AFIT/GE/ENG/91D-15

A WALSH-DOMAIN ADAPTIVE FILTER

THESIS

Larry J. Duvall
Captain, USAF

AFIT/GE/ENG/91D-15

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
A Walsh-domain Adaptive Filter

5. FUNDING NUMBERS

6. AUTHOR(S)
Larry J. Duvall, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/91D-15

9. SPONSORING, MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING, MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Two Walsh-domain dyadic convolution adaptive filters are developed using a circular convolution frequency-domain filter (FDF1) and the Fast LMS adaptive filter (FDF2): WDF1 and WDF2 respectively. General theory of time-domain adaptive filters and a theoretical analysis of the FDF1, FDF2, WDF1, and WDF2 filters are presented. WDF1 and WDF2 software implementations are shown to be error free. A time-domain filter (TDF) and a FDF2 frequency-domain filter (FDF) are implemented for comparison testing. The WDF1, WDF2, TDF, and FDF filters are tested using time-shifted sinusoidal and rectangular noisy and noiseless signals. WDF1 and WDF2 are shown to converge faster and produce less error filtering discontinuous signals, relative to the TDF and FDF performance. WDF1 and WDF2 are shown to converge slower and produce more error filtering continuous signals, relative to TDF and FDF performance. WDF1 is shown to perform better for noiseless signals, relative to WDF2 performance. WDF2 is shown to perform better for noisy signals, relative to WDF1 performance. WDF1 and WDF2 filtering performance was shown to degrade with increasing time shift. A processing speed comparison showed WDF1 to be faster than the TDF, FDF, and WDF2 filters.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Adaptive Filter, Walsh Transform, Least Mean Square	273
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

AFIT/GE/ENG/91D-15

A WALSH-DOMAIN ADAPTIVE FILTER

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Larry J. Duvall, B.S.E.E.
Captain, USAF

December 1991

Preface

The purpose of this study was to investigate the development of a Walsh-domain adaptive filter. Discrete Walsh-domain signal processing provides computational and hardware simplification versus discrete frequency-domain signal processing requirements. Therefore, development of a Walsh-domain adaptive filter would provide a robust self-designing signal processor suitable for applications which limit available power and space, particularly space-based signal processing. By extending a circular convolution frequency-domain adaptive filter and the Fast LMS frequency-domain adaptive filter into the Walsh-domain, two Walsh-domain adaptive filters were developed.

I can not fully express my appreciation of the invaluable guidance provided by my thesis advisor, Captain Rob Williams. His helpful suggestions were not mere cookbook solutions, they provided this student with exciting opportunities for discovery and insight. I would also like to thank those on the thesis committee; Dr Matthew Kabrisky and Martin Desimio.

Above all, I thank my best friend and wife, Shannon. Any accomplishments that I have made were gained by her sacrifices.

Larry J. Duvall

Table of Contents

List of Figures

List of Tables

Abstract

Two Walsh-domain dyadic convolution adaptive filters are developed using a circular convolution frequency-domain filter (FDF1) and the Fast LMS adaptive filter (FDF2): WDF1 and WDF2 respectively.

General theory of time-domain adaptive filters and a theoretical analysis of the FDF1, FDF2, WDF1, and WDF2 filters are presented. WDF1 and WDF2 software implementations are shown to be error free. A time-domain filter (TDF) and a FDF2 frequency-domain filter (FDF) are implemented for comparison purposes. The WDF1, WDF2, TDF, and FDF filters are tested using time-shifted sinusoidal and rectangular noisy and noiseless signals. WDF1 and WDF2 are shown to converge faster and produce less error filtering discontinuous signals, relative to the TDF and FDF performance. WDF1 and WDF2 are shown to converge slower and produce more error filtering continuous signals, relative to TDF and FDF performance. WDF1 is shown to perform better for noiseless signals, relative to WDF2 performance. WDF2 is shown to perform better for noisy signals, relative to WDF1 performance. WDF1 and WDF2 filtering performance was shown to degrade with increasing time shift. A processing speed comparison showed WDF1 to be faster than the TDF, FDF, and WDF2 filters.

A WALSH-DOMAIN ADAPTIVE FILTER

I. Introduction

1.1 Background

The development of space-based adaptive signal processors is an active area of research that has immediate military potential. In developing space-based systems, system robustness is a desirable quality. Adaptive systems exhibit that quality [11]. Therefore, "an adaptive system that continually seeks the optimum within an allowed class of possibilities, using an orderly search process, would give superior performance compared with a system of fixed design" [9:5]. Real-time signal processing requires that the search process be conducted using a minimum of time, which is why for example, frequency-domain adaptive signal processors are more commonly used than time-domain processors. Speed in this case is costly because hardware required to implement the Fast Fourier Transform (FFT) is sophisticated and expensive. Using other transform domains could possibly afford a hardware savings and retain or surpass the processing speed of frequency-domain processing. This thesis looks at the development of a Walsh-domain adaptive filter.

1.1.1 The Walsh-Domain. Signal processing of stochastic signals can be accomplished in a variety of domains, the most common being time and frequency. There are basically two reasons for choosing one processing domain versus another. First, a computational advantage may be achieved. For example, the simplicity achieved in performing the convolution of two discretely sampled data sets with the Fast Fourier Transform (FFT) versus performing the convolution sum [10]. The second, information of interest may be more visible in one domain than another [10]. Frequency distribution is clearly indicated in the frequency-domain, while amplitude characteristics are clearly indicated in the time-domain.

Walsh-domain processing concerns the representation of a signal as a sum of weighted, bipolar-pulse waveforms. In the Walsh-domain, continuous and discrete time signals are represented in terms of an orthonormal set of basis functions consisting of waveforms with discrete ± 1 amplitude values [1:17–18]. Walsh functions are defined over a finite time interval,

which is usually normalized to one, and by an ordering number n [1:9-10] which specifies the number of times the function passes thru zero over the time interval; referred to as *sequency* [1:15].

The Discrete Walsh Transform (DWT) of a discretely sampled continuous time signal is accomplished generally in the same manner as the Discrete Fourier Transform (DFT). The N-point DWT transform pair [1:50] is as follows:

$$X_n = 1/N \sum_{i=0}^{N-1} x_i WAL(n,i) \tag{1.1}$$

and

$$x_i = \sum_{n=0}^{N-1} X_n WAL(n,i) \tag{1.2}$$

Figure 1.1 shows the $N = 8$ series Discrete Walsh functions. The DWT spectral components are represented by the X_n terms and n is the ordering number. The data sequence is indicated by x_i, where i is the discrete time index and N represents the number of data values being transformed.

The kernel of the DWT sum, $WAL(n,i)$, is ± 1 depending upon the values of n and i. Equations 1.1 and 1.2 show that the DWT produces its own inverse. Just as in the case of the DFT, algorithms exist to produce a faster transform; the Fast Walsh Transform (FWT) [1:58].

The DWT should be considered for time-limited waveforms, because "...it is possible for the power spectrum to be sequency limited although the corresponding time functions are time limited" [1:103]. For a time-limited waveform, the DWT sequency spectra has a finite number of terms while the DFT would generate an infinite frequency spectra [1:103] such that, in general, "a continuous type of waveform favours using the DFT and a discontinuous type of waveform favours using the DWT" [1:132].

In terms of an implementation comparison with the DFT, the DWT is much simpler and faster because the DWT transformation matrix is composed of real values (± 1) while the DFT transformation matrix is composed of complex values [1:51]. Also, the DWT requires no multiplication operations because the transform matrix values are ± 1, thus reducing the multiplication operations to simply addition.

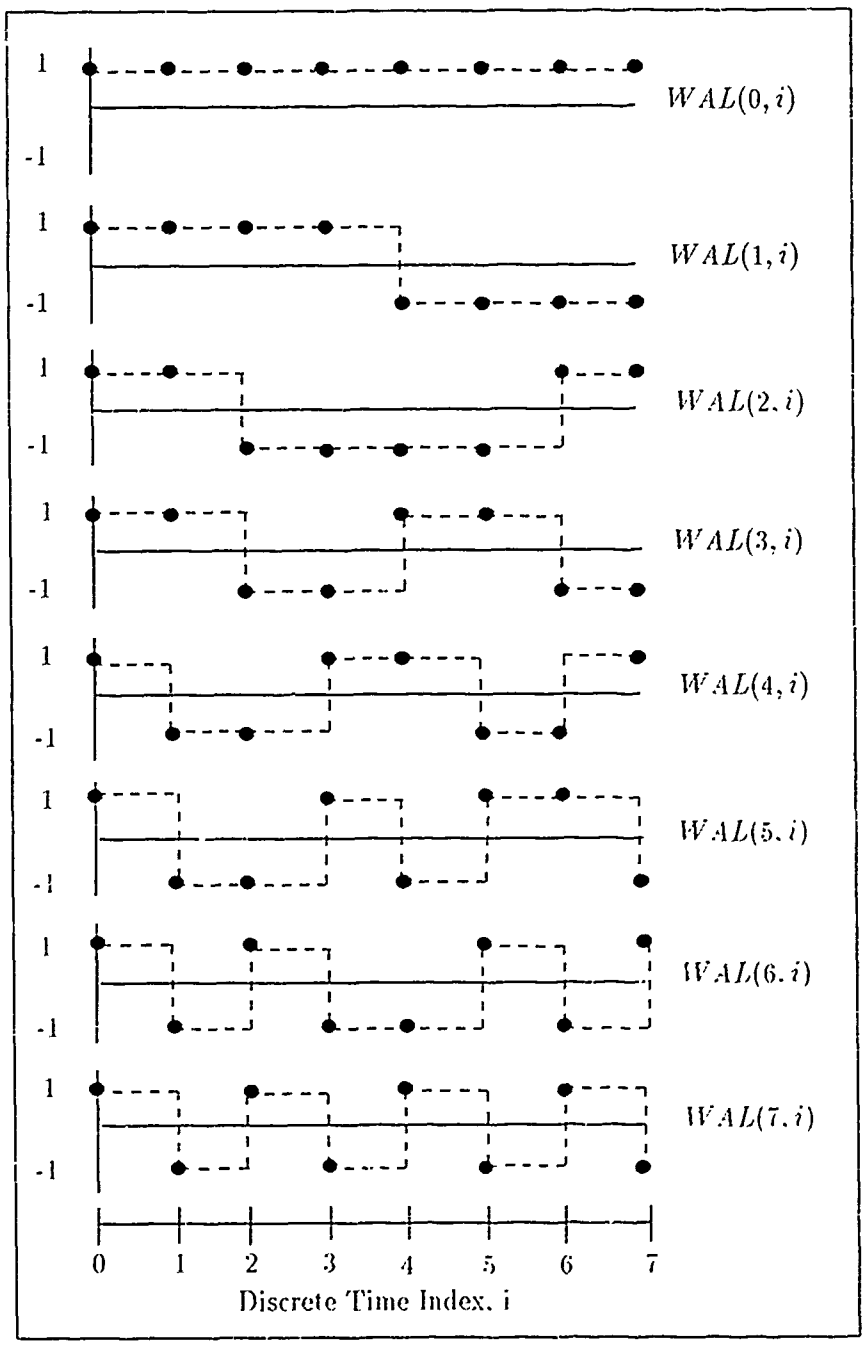

Figure 1.1. Discrete Walsh Functions for $N = 8$, in sequency order.

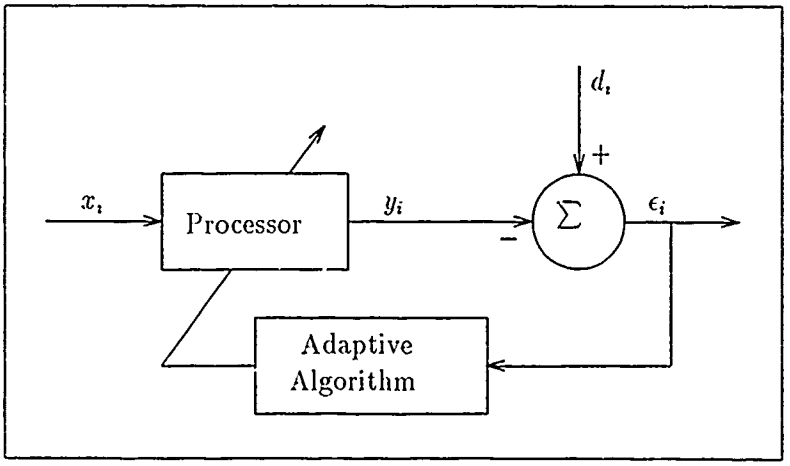

Figure 1.2. Adaptive Filter

1.1.2 Adaptive Filters. The adaptive filter is a signal processor that possesses self-adjusting capability and is time varying [3:2-14]. The most common form of adaptive filter uses a closed-loop design (See Figure 1.2). The filter processes the input x_i to yield an output y_i which is compared against a desired signal d_i, yielding an error signal ϵ_i. The desired signal d_i serves as a "*training signal*" for the filter [9:18]. Adaptive filters can be implemented as time domain filters or as transform-based filters, which are referred to as block processing adaptive filters.

1.1.2.1 Time-Domain Adaptive Filters. The single-input time-domain LMS adaptive filter processor structure used in this thesis is referred to as an Adaptive Transversal Filter (ATF) [9:16]. As shown in Figure 1.3, the ATF is a standard Finite Impulse Response (FIR) digital filter structure with tap weights which adapt with time per a predefined adaptation algorithm. In this figure , the ATF has $(L + 1)$ taps. Since the filter weights are adjusted using the Widrow-Hoff LMS algorithm, $h_j(i)$ represents the tap value of the jth weight at time i [9:100].

1.1.2.2 Block-Processing Adaptive Filters. Since the DWT is generated in a manner very similar to the DFT, frequency-domain block-processing filter theory might provide insight into how to develop a Walsh-domain block-processing filter. This section looks at a Frequency-domain block-processing filter that performs circular convolution.

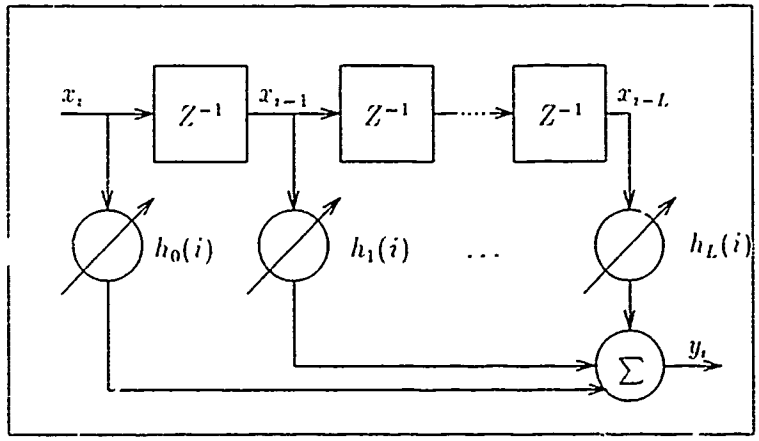

Figure 1.3. Adaptive Transversal Filter

One significant difference between a transform-based adaptive filter and the time-domain filters is that the data is processed in blocks [2:146] (Figure 1.4). The constant N specifies the block size. The N data points of d_t and x_t for the kth block are transformed using the Fast Fourier Transform (FFT). The transform components of the kth input block, $X_n(k)$, are then multiplied with the respective frequency-domain filter weights, $H_n(k)$, for the kth block to produce the kth block output spectral components, $Y_n(k)$. The $Y_n(k)$ components are then subtracted from the corresponding block of desired signal spectral components, $D_n(k)$, to produce the error components, $E_n(k)$. The $E_n(k)$ values are used to update the filter weights, $H_n(k)$.

In all cases, the index n indicates the spectral component, k indicates the block being processed, and i is the discrete-time index. Because the FFT of a discrete-time signal results in complex values, the frequency-domain nth bin $H_n(k)$ and $E_n(k)$ components are complex. Unlike time-domain adaptive filters which update the weights at each i, block processing filters (BPF) update once per block. The complex LMS algorithm is the most popular weight update algorithm [2:147].

1.2 Problem Statement

Merging DWT theory and transform-based adaptive filter theory would enable the development of robust self designing signal processors which possess minimal computational requirements suitable for applications which limit available power and space, particularly space-based signal processing [11].

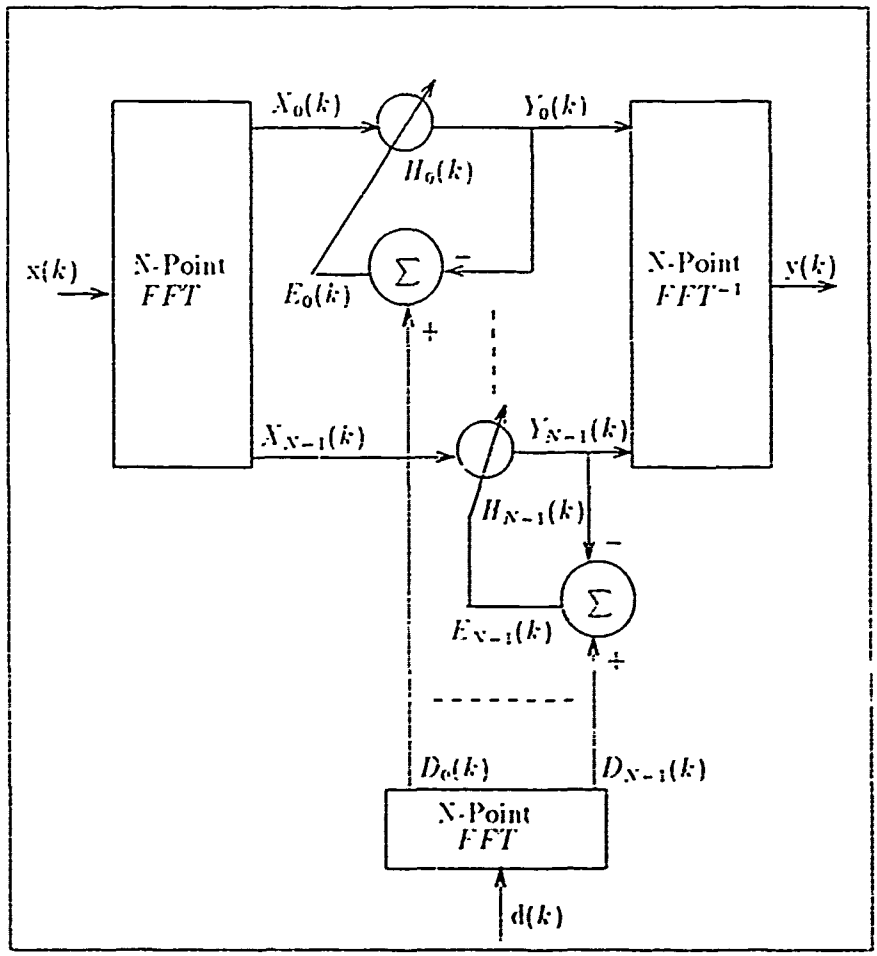

Figure 1.4. A Frequency-Domain Adaptive Filter

1.3 Research Objective

The primary objective of this thesis is to develop working frequency-domain and Walsh-domain block-processing adaptive filters in software and categorize their performance in terms of convergence speed, output error, and processing speed.

1.4 Review of Literature

This section presents a brief discussion of the published research that pertains to the subject of this thesis. Two research efforts were found in the literature pertaining to Walsh-domain adaptive filters. The first concerned the hardware implementation of a Walsh-domain adaptive filter. The second presented an Adaptive Walsh equaliser. The following sections address each of the two research efforts.

1.4.1 Walsh-Domain Adaptive Filter. Literature research revealed work done in Walsh-domain adaptive filtering done by American Electronic Laboratories [8]. The Walsh Adaptive Filter developed by American Electronics Laboratories was designed to adaptively filter pulsed waveforms of varying widths, frequency, and time of arrival. The adaptive nature of the filter, however, was in relation to the threshold used and was not LMS in nature. This was not a true Walsh-domain adaptive filter since the emphasis was on threshold but is worth mentioning since, as far as could be determined, it is the only combined implementation of Walsh Transform and adaptive filtering theory.

1.4.2 Adaptive Walsh Equaliser. The Adaptive Walsh Equaliser (AWE) [4] is a continuous-time filter that performs linear convolution using a weighted sum of M variable coefficient continuous-time Walsh functions. Applying the superposition principle, the continuous-time input signal is convolved with each of the orthogonal impulse response Walsh components and the individual responses are summed to produce the output, which is compared to a desired signal. The impulse response component weights are adapatively adjusted to minimize the output error. This filter operates in the time-domain, while this thesis investigates operation in the sequency-domain.

1.5 Assumptions

The Walsh-domain and frequency-domain filters will be implemented in software and it is assumed that noise contributed by round-off errors and quantization effects due to

finite word lengths is insignificant. Errors of this type are generally ignored since their noise contributions can be minimized with FIR filter implementations [12:7].

1.6 Scope

The research will be limited to the development of frequency-domain and Walsh-domain adaptive filters using the Fast LMS and Circular Convolution frequency-domain adaptive filter structures [2:146–157]. A performance comparison of the filters will be made with a time-domain LMS adaptive filter based on convergence speed, error performance, and processing speed. The signal sets used are time-shifted noisy and noiseless sinusoidal and rectangular signals.

1.7 Hardware Requirements

This thesis requires a PC/AT-class computer with at least a 512K Ram Disk, 640K of base memory, and a single floppy drive. No additional hardware is required.

1.8 Software Requirements

All software programs developed in this thesis were implemented using Turbo Pascal version 6.0.

1.9 Approach and Presentation

The approach used in this research includes a literature search, software implementation of the Walsh and frequency-domain adaptive filters, software testing, and a performance comparison of the filters. The remaining chapters are structured as follows:

- Chapter II presents a theoretical analysis of Discrete Walsh Transform theory and the frequency-domain filters.
- Chapter III presents two Walsh-domain adaptive filters developed by extending the frequency-domain filter designs into the Walsh-domain.
- Chapter IV discusses the Walsh-domain adaptive filter implementations in software and the tests used to verify integrity.
- Chapter V discusses the performance comparison results.
- Chapter VI presents the conclusions and recommendations.

1.10 Original Research Contributions

To the author's knowledge, the unique and significant contributions presented in this research include:

1. Merging of Discrete Walsh Transform (DWT) theory and adaptive filter theory.
2. An extension of a frequency-domain circular convolution filter design to a Walsh-domain dyadic convolution filter design, Walsh-domain Filter 1 (WDF1), that provides a processing speed and discontinuous input signal filtering improvement over the Fast LMS and time-domain LMS filters.
3. Development of a new transform-domain LMS algorithm, Walsh Transform LMS algorithm 1 (WLMS1), which permits use of the DWT in a frequency-domain filter that performs circular convolution.
4. A new Walsh-domain filter design, Walsh-domain Filter 2 (WDF2), for a Walsh-domain block processing filter that uses a modified form of the "overlap-save" method, has improved noisy input error performance over the WDF1 filter, and provides processing speed and discontinuous input signal filtering improvement over the Fast LMS filter.

II. Background

This chapter presents the Walsh-domain and adaptive filter theory used in the development of a Walsh-domain adaptive filter. Adaptive filter theory is presented in two areas: time-domain adaptive filters and block processing adaptive filters. The block processing adaptive filter discussion presents the Fast LMS frequency-domain filter and a circular convolution frequency-domain filter. Both filters are used in Chapter 3 as templates in the design of a Walsh-domain filter.

2.1 Walsh-Domain Theory.

Signal processing of stochastic signals can be accomplished in a variety of domains, the most commonly used being time and frequency. There are basically two reasons for choosing one processing domain versus another. First, a computational advantage may be achieved [10]. One example is the simplicity achieved in performing the convolution of two discretely sampled data sets with the Fast Fourier Transform (FFT) versus performing the convolution sum. The second, information of interest may be more visible in one domain than another [10]. Power distribution is clearly indicated in the frequency-domain, whereas amplitude characteristics are clearly indicated in the time-domain.

2.1.1 Walsh Functions. Walsh Domain processing represents the signal as a sum of weighted, bipolar pulse waveforms. In the Walsh-domain, continuous and discrete signals are represented in terms of an orthonormal set of basis functions consisting of waveforms with discrete ± 1 amplitude values [1:17-18]. Walsh functions are defined over a finite time interval, which is usually normalized to one, and by an ordering number n [1:9-10]. The ordering number specifies the number of times the function passes thru zero over the time interval, and is referred to as *sequency* [1:15]. Discrete Walsh functions are determined by sampling the corresponding continuous Walsh function at N equally spaced points over the interval (0,1). The number of samples, N, must be a power of 2 [1:50]. For a series of $N = 2^p$ terms, the discrete Walsh functions can be specified as [1:59]

$$\begin{aligned} WAL(n,i) &= \prod_{r=0}^{p-1} (-1)^{n_{p-1-r}(i_r + i_{r+1})} \\ i,n &= 0,1,2,\ldots,N-1 \\ r &= 0,1,2,\ldots,p-1 \end{aligned} \tag{2.1}$$

The indexes i, n are expressed in terms of their binary digits such that

$$\begin{aligned} i &= (i_p\ i_{p-1}\ \ldots\ i_1\ i_0)_2 \\ n &= (n_p\ n_{p-1}\ \ldots\ n_1\ n_0)_2 \end{aligned} \tag{2.2}$$

where the subscript 2 on the right side of the equality in Equation 2.2 indicates the binary representation of the left side of the equality. The leftmost bit is the most significant. An example $WAL(n, i)$ term calculation and the first 8 discrete Walsh functions are presented in Section A.1.

2.1.2 Discrete Walsh Transform. The Discrete Walsh Transform (DWT) of a discretely sampled continuous time signal is accomplished generally in the same manner as the Discrete Fourier Transform (DFT). The discrete Walsh transform pair [1:50] is as follows:

$$X_n = 1/N \sum_{i=0}^{N-1} x_i WAL(n, i) \tag{2.3}$$

and

$$x_i = \sum_{n=0}^{N-1} X_n WAL(n, i) \tag{2.4}$$

The DWT spectral components are represented by the X_n terms and n is the ordering number. The data sequence is indicated by x_i, where i is the discrete time index while N represents the number of data values being transformed. The N input data values being transformed are not assumed to be periodic. The kernel of the sum, $WAL(n, i)$, is ± 1 depending upon the values of n and i (Equation 2.1). Because the transform kernel is ± 1 the DWT requires no multiplies while conversely, the DFT requires N^2 complex multiplies. Clearly, the DWT has a computational advantage over the DFT. As in the case of the DFT, Fast Walsh Transform (FWT) algorithms exist [1:58–74] [7]. The transform is linear [1:50] so that if

$$x_i \overset{W}{\longleftrightarrow} X_n \tag{2.5}$$

and

$$y_i \overset{W}{\longleftrightarrow} Y_n \tag{2.6}$$

then

$$ax_i + by_i \overset{W}{\longleftrightarrow} aX_n + bY_n \tag{2.7}$$

where a and b are real constants and W represents the DWT operator.

2.1.2.1 Walsh Matrix. The DWT can also be represented as a vector matrix multiplication operation so that

$$X_n = (1/N)W_N \cdot x \tag{2.8}$$

where x is the $N \times 1$ data sequence vector, X_n is the $N \times 1$ spectral component vector, and W_N is the $N \times N$ Walsh matrix. The matrix W_N and the input vector x are defined

$$W_N = \begin{bmatrix} WAL(0,0) & WAL(0,1) & \dots & WAL(0,N-1) \\ WAL(1,0) & WAL(1,1) & \dots & WAL(1,N-1) \\ \vdots & \vdots & & \vdots \\ WAL(N-1,0) & WAL(N-1,1) & \dots & WAL(N-1,N-1) \end{bmatrix} \tag{2.9}$$

and

$$x = [x_0 \ x_1 \ \dots \ x_{N-1}] \tag{2.10}$$

The rows of W_N are the first N Walsh functions, where the sequency 0 function is the first row, and the matrix is diagonally symmetric. The first $N/2$ columns are characterized by the fact that, in numbering the first row 0, the even and consecutive odd numbered rows are equal: for example row0=row1 and row2=row3. This characteristic will be referred to as row symmetry.

The significance of the row symmetry characteristic can be seen when transforming zero end-padded sequences. An $N/2$ zero end-padded input data vector produces a spectrum where, starting with the 0 spectrum component, the even numbered components are equal to their consecutive odd numbered components: for example $X_0 = X_1$ and $X_2 = X_3$. Therefore, the resulting spectrum is component symmetric. Example Discrete Walsh Transforms of zero end-padded vectors are presented in Section A.2.

2.1.2.2 Spectrum Characteristics. "An important feature of the power spectrum using Walsh functions is that it is possible for the power spectrum to be sequency limited although the corresponding time functions are time limited" [1:103]. In comparison, the corresponding frequency spectrum for time-limited waveforms cannot be frequency limited. The representation of a continuous waveform with Walsh functions results in a more complex spectrum than that produced using Fourier analysis [1:103]. Conversely, the

representation of discontinuous waveforms using Fourier analysis results in a more complex spectrum than that produced using Walsh functions.

Simple rectangular and sinusoidal waveforms are examples of discontinuous and continuous waveforms, respectively. In the case of periodic signals, the DWT of one period of the signal produces a different sequency spectrum than the DWT of two periods. This characteristic is demonstrated in Section A.3 for a simple sinusoid and rectangular signal. The sum of the sequency components squared equals the input signal power.

2.1.3 Circular Time Shift Effects. Unlike the DFT, the DWT does not possess a circular shift property [1:51]. Given a sequence x_i which produces a DFT X_k, the circularly shifted sequence x_{i-m} produces a DFT of $exp^{-j(2\pi k/N)m}X_k$. The spectrum of the circularly shifted sequence retains the same magnitude characteristics as the spectrum of the unshifted sequence but possesses a different phase characteristic. Therefore, the DFT is referred to as a shift invariant transform. The shift variant nature of the DWT is demonstrated in Section A.3 for a simple sinusoid and rectangular signal.

2.1.4 Convolution. The linear convolution of two discrete N point sequences x_i and y_i in the time domain is defined as

$$z_\tau = \sum_{i=0}^{N-1} x_i y_{\tau-i} \tag{2.11}$$

The convolution theorem for the Fourier Transform states that the convolution of two N-point time series x_i and y_i can be performed by multiplying the Fourier Transforms of the two series and taking the inverse transform of the product. This results in circular convolution due to the fact that the DFT assumes the sequences to be periodic. Therefore it is necessary to pad the sequences with N zeros, transform the 2N-point series, multiply the 2N-point DFTs of the two sequences, and inverse transform the result to perform linear convolution in the time domain.

In the case of the DWT, no relationship exists with regard to performing linear time domain convolution. This is demonstrated by replacing x_i and $y_{\tau-i}$ in Equation 2.11 with their equivalent inverse Discrete Walsh Transform. Performing this substitution gives [1:99]

$$z_\tau = \sum_{i=0}^{N-1}[\sum_{n=0}^{N-1} X_n WAL(n,i)][\sum_{l=0}^{N-1} Y_l WAL(l,\tau-i)] \tag{2.12}$$

$$= \sum_{n=0}^{N-1} \sum_{l=0}^{N-1} X_n Y_l [\sum_{i=0}^{N-1} WAL(k,i) WAL(l,\tau - i)] \tag{2.13}$$

The bracketed sum term represents the convolution of the discrete Walsh functions, and demonstrates the fact that linear time convolution is not achieved. What is achieved, is referred to as *dyadic convolution.*

The Walsh addition theorem [1:53] states the following relationship

$$WAL(n,i)WAL(m,i) = WAL(n \oplus m, i) \tag{2.14}$$

where $\oplus$ indicates modulo-2 addition for the binary representations of n and m. Dyadic convolution is defined as [1:100]

$$\begin{aligned} z_\tau &= 1/N \sum_{i=0}^{N-1} x_i y_{\tau \oplus i} \\ &= x_i \star y_i \end{aligned} \tag{2.15}$$

and substituting the Discrete Walsh Transform (DWT) expression for $y_{\tau \oplus i}$ into Equation 2.15 produces [1:100]

$$z_\tau = 1/N \sum_{i=0}^{N-1} x_i [\sum_{n=0}^{N-1} Y_n WAL(n, \tau \oplus i)] \tag{2.16}$$

Applying the addition theorem (Equation 2.14 to Equation 2.16 produces [1:100]

$$\begin{aligned} z_\tau &= 1/N \sum_{n=0}^{N-1} Y_n \sum_{i=0}^{N-1} x_i WAL(n,i) WAL(n,\tau) \\ &= \sum_{n=0}^{N-1} X_n Y_n WAL(n,\tau) \end{aligned} \tag{2.17}$$

which establishes the relationship

$$x_i \star y_i \overset{W}{\longleftrightarrow} X_n Y_n \tag{2.18}$$

where W is the DWT operator. In comparison, this realtionship demonstrates that distinct sets of relationships exist for the Walsh and Fourier series. Each establishes a form of

convolution theory with the difference being that the Fourier version utilizes arithmetic addition for the recursive time shift and the Walsh version utilizes modulo-2 addition.

Dyadic convolution is similar to circular convolution in that there are N product terms associated with the result at each τ shift when convolving two N point sequences. Linear convolution produces $\tau + 1$ product terms for each τ shift. An example is presented in Section A.2 that demonstrates the relationship stated in Equation 2.18.

2.1.5 Correlation. Discrete autocorrelation in real time is defined by [1:100]

$$R_\tau = 1/N \sum_{i=0}^{N-1} x_i x_{i+\tau} \tag{2.19}$$

where $\tau = 0, 1, 2, \ldots, m$ and $m \ll N$. The constant m represents the total correlation lag. Discrete autocorrelation in dyadic time is defined by [1:100]

$$R_\tau = 1/N \sum_{i=0}^{N-1} x_i x_{i \oplus \tau} \tag{2.20}$$

Since modulo-2 addition and subtraction are identical operations, dyadic convolution and correlation produce the same result [1:101]. Correlation of x_i and y_i can be accomplished by multiplying the x_i sequence DFT conjugate and the y_i sequence DFT. This is shown by applying the convolution theorem for the Fourier series and the relationship

$$x_{-i} \overset{\mathcal{F}}{\longleftrightarrow} X_k^* \tag{2.21}$$

where $\mathcal{F}$ is the DFT operator. The inverse DFT produces the cross-correlation of x_i and y_i. No such relationship exists for DWTs since the DWT components are real.

2.2 Adaptive Filters.

An adaptive filter is a signal processor that possesses self adjusting capability and is time varying [3:2-11]. The filter is continually modifying its current state in response to input and output signals [3:2-11]. The most common form of adaptive filter used in signal processing is the closed-loop, which is illustrated in Figure 2.1. This filter processes the input x_i to yield an output y_i which is compared against a desired signal d_i, yielding an error signal ϵ_i. The desired signal, d_i, serves as a "*training signal*" for the filter [9:18]. All three variables

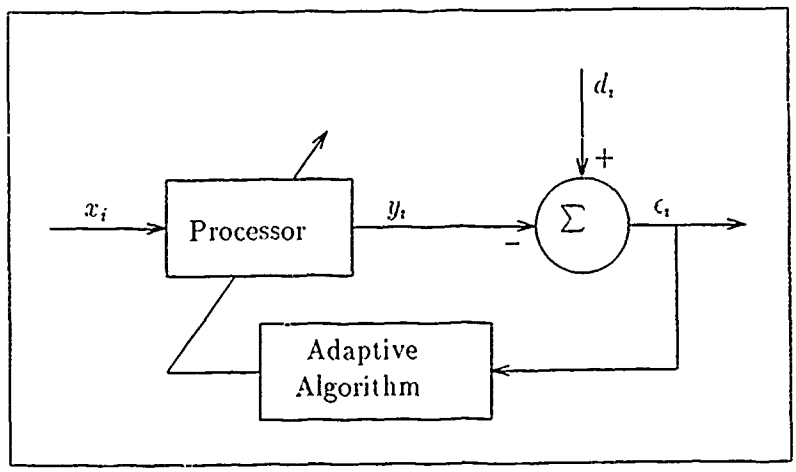

Figure 2.1. Adaptive Filter Block Diagram

represent discrete values and the subscript i represents the discrete time index. Adaptive filters can be implemented as time domain filters or as transform based filters, also referred to as block processing adaptive filters.

2.2.1 Time-domain Adaptive Filters. The single-input time-domain LMS adaptive filter processor structure used in this thesis is referred to as an Adaptive Transversal Filter [9:9]. As shown in Figure 2.2, the Adaptive Transversal Filter is essentially a Finite Impulse Response (FIR) digital filter with adaptive weights that are represented such that the jth tap at time i is $h_j(i)$, where $j = 0, 1, \ldots, L$. Assuming a causal filter, the filter size is $L + 1$. Weight update at time i is most commonly performed using the Widrow-Hoff LMS algorithm [9:100]. Filter weight values are collectively represented as a weight vector which is defined as follows:

$$\mathrm{h}(i) = [h_0(i) \;\; h_1(i) \;\; \ldots \;\; h_L(i)]^T \tag{2.22}$$

The filter output at time i is produced from the linear combination of products formed by multiplying the filter weights with their corresponding delayed input signal values as specified by [9:17]

$$\begin{aligned} y_i &= \sum_{l=0}^{L} x_{i-l} h_l(i) \\ &= \mathrm{x}^T(i)\mathrm{h}(i) \end{aligned} \tag{2.23}$$

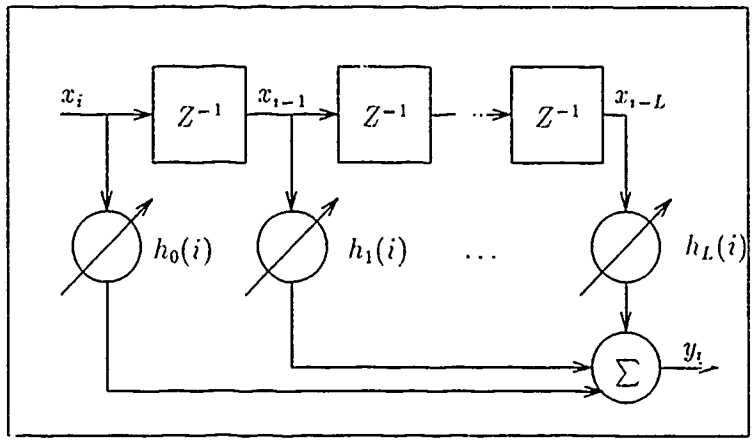

Figure 2.2. Adaptive Transversal Filter

where

$$\mathrm{x}(i) = [x_i \;\; x_{i-1} \;\; \ldots \;\; x_{i-L}]^T \tag{2.24}$$

The error signal is specified by [9:19]

$$\begin{aligned} \epsilon_i &= d_i - y_i \\ &= d_i - \mathrm{x}^T(i)\mathrm{h}(i) \end{aligned} \tag{2.25}$$

As a measure of optimal performance, the LMS adaptive filter seeks to minimize the mean squared error $E[\epsilon_i^2]$; with the ideal case typically being $E[\epsilon_i^2] = 0$ [3:2-16]. The mean squared error is defined as follows [3:2-16]:

$$\begin{aligned} E[\epsilon_i^2] &= E[d_i^2 + \mathrm{h}^T(i)\mathrm{x}(i)\mathrm{x}^T(i)\mathrm{h}(i) - 2d_i\mathrm{x}^T(i)\mathrm{h}(i)] \\ &= E[d_i^2] + E[\mathrm{h}^T(i)]E[\mathrm{x}(i)\mathrm{x}^T(i)]E[\mathrm{h}(i)] - 2E[d_i\mathrm{x}^T(i)]E[\mathrm{h}(i)] \\ &= E[d_i^2] + \mathrm{h}^T(i)E[\mathrm{x}(i)\mathrm{x}^T(i)]\mathrm{h}(i) - 2E[d_i\mathrm{x}^T(i)]\mathrm{h}(i) \end{aligned} \tag{2.26}$$

There are two key assumptions made in the derivation of Equation 2.26. First, in progressing from the first to the second line, the weight vector and the input signal vector are assumed uncorrelated. The second assumption is that the weight vector converges to a solution. When the weight vector converges $E[\mathrm{h}(i+1) - \mathrm{h}(i)]$ equals the zero vector and the weight vector can be treated as a constant. This leads to the final form of Equation 2.26.

which is [9:20]

$$E[\epsilon_i^2] = E[d_i^2] + \mathrm{h}^T R\mathrm{h} - 2P^T\mathrm{h} \tag{2.27}$$

where R is the autocorrelation matrix and is defined as [3:2-17]

$$R = E[\mathrm{x}(i)\mathrm{x}^T(i)] = \begin{bmatrix} \phi_{xx}(0) & \phi_{xx}(-1) & \dots & \phi_{xx}(-L) \\ \phi_{xx}(1) & \phi_{xx}(0) & \dots & \phi_{xx}(1-L) \\ \phi_{xx}(2) & \phi_{xx}(-1) & \dots & \phi_{xx}(2-L) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_{xx}(L) & \phi_{xx}(L-1) & \dots & \phi_{xx}(0) \end{bmatrix} \tag{2.28}$$

and $\phi_{xx}(n) = E[x_i x_{i+n}]$. P is the cross correlation vector

$$P = E[d_i\mathrm{x}(i)] = \begin{bmatrix} \phi_{dx}(0) \\ \phi_{dx}(-1) \\ \vdots \\ \phi_{dx}(-L) \end{bmatrix} \tag{2.29}$$

where $\phi_{dx}(n) = E[d_i x_{i+n}]$. Under the assumption that the input signal and desired signal are stationary, R and P are constant and as such require no time subscript.

Equation 2.27 is referred to as the performance equation or mean-square error(MSE) equation for the LMS filter. Equation 2.27 characterizes a performance surface which the filter searches to find the global minimum which is guaranteed to exist by the quadratic nature of the equation. The weight vector that satisfies the global minimum is referred to as h_{opt}. Solving for h_{opt} requires taking the gradient of $E[\epsilon_i^2]$, and setting the gradient equal to the zero vector. The gradient is being taken with respect to the weight vector W. Defining the gradient vector ∇_w as [9:2-18]

$$\nabla_w F = [\partial F/\partial w_0 \;\; \partial F/\partial w_1 \;\; \dots \;\; \partial F/\partial w_L]^T \tag{2.30}$$

where F represents a function of w and ∇_w the gradient operator. Applying the ∇_w operator to Equation 2.27 and solving for the minimum produces [9:21]:

$$\begin{aligned} 0 &= \nabla_w E[d_i] + \nabla_w \mathrm{h}^T R\mathrm{h} - \nabla_w 2P^T\mathrm{h} \\ &= 0 + \nabla_w(\mathrm{h}^T)(R\mathrm{h}) - \nabla_w 2P^T\mathrm{h} \\ &= 2R\mathrm{h} - 2P \end{aligned} \tag{2.31}$$

where the left side of the equality is the zero vector. Evaluating Equation 2.31 for h yields h_{opt} such that [9:22]

$$\mathrm{h}_{opt} = R^{-1}P \tag{2.32}$$

An important footnote to this result is that this assumes R to be invertible [9:2-18]. Substituting Equation 2.32 into Equation 2.31 for h yields the minimum mean-square error, which is specified as [9:22]

$$\xi_{min} = E[d_i^2] - P^T R^{-1} P \tag{2.33}$$

Equations 2.32 and 2.33 clearly show that the optimum filter solution and the optimum MSE performance depends on the autocorrelation and crosscorrelation statistics. For an input signal that is statistically wide sense stationary, the R matrix can be evaluated and will be composed of constants. Evaluation of the P vector is accomplished by cross correlating $\mathrm{x}(i)$ and d_i. The performance surface defined by Equation 2.27 as mentioned earlier is quadratic in nature.

The gradient at any point on the performance surface corresponds proportionately to the surface slope. The rate with which the weight vector converges also is proportional to the gradient and therefore the surface slope.

A popular measure of adaptive filter performance is the *learning curve* which measures the MSE as a function of time [10]. Generally, $E[\epsilon_i^2]$ is estimated by ensemble averaging ϵ_i^2 versus i over a number of individual runs of the input. The minimum MSE or ξ_{min} occurs when the filter taps have reached their optimum solution as defined in Equation 2.32. In most cases, an exact match with the desired signal is not achieved due to adaptation noise in the filter tap update and noise in the filter input x_i [10].

The filter weights for each tap position are updated using the LMS algorithm:

$$\mathrm{h}(i+1) = \mathrm{h}(i) + 2\mu\epsilon_i\mathrm{x}(i) \tag{2.34}$$

after each time instant, based upon the error difference ϵ_i between the filter output y_i and the desired signal d_i. The constant μ is referred to as the gain constant and is used to adjust adaptation speed and control the stability of adaptation [9:100]. The filter searches the performance surface for the global minimum using the LMS algorithm. As the input data is processed, the LMS algorithm uses a gradient estimate derived from the instantaneous error to search the Performance Surface [9:99-100]. As stated in Equation 2.34, the next weight vector, $h(i+1)$, is calculated by adjusting the previous weight vector by the scaled product of the instantaneous error and input vector. In the case of nonstationary signals, if the statistics vary slowly, the gain constant μ can be increased to allow the filter to track the nonstationarity at the expense of increased adaptation noise. For rapidly changing input signal statistics, the filter will not converge to an optimum weight solution [3:2-21].

2.2.2 Block Processing Adaptive Filters. This section presents the Fast LMS adaptive filter and a circular convolution frequency-domain filter. For each filter, the presentation is as follows:

1. Time-domain input vector definition.
2. Frequency-domain input.
3. Output Calculation.
4. Frequency-domain weight update.
5. Time-domain representation.
6. Optimum weight vector.
7. Computational requirements in terms of multiplications.

2.2.2.1 Circular Convolution Model One significant difference in the implementation of a transform based adaptive filter versus a time-domain filter is that the data is processed in blocks [2:146]. A frequency domain adaptive filter that performs circular convolution is depicted in Figure 2.3 [2:147] and will be referred to in the remainder of this thesis as Frequency-dom.in Filter 1 (FDF1). The block processing nature of this filter is reflected in Figure 2.3 using the notation $x(k)$, $d(k)$, and $y(k)$, which represent the kth block input vector, desired vector, and output vector respectively.

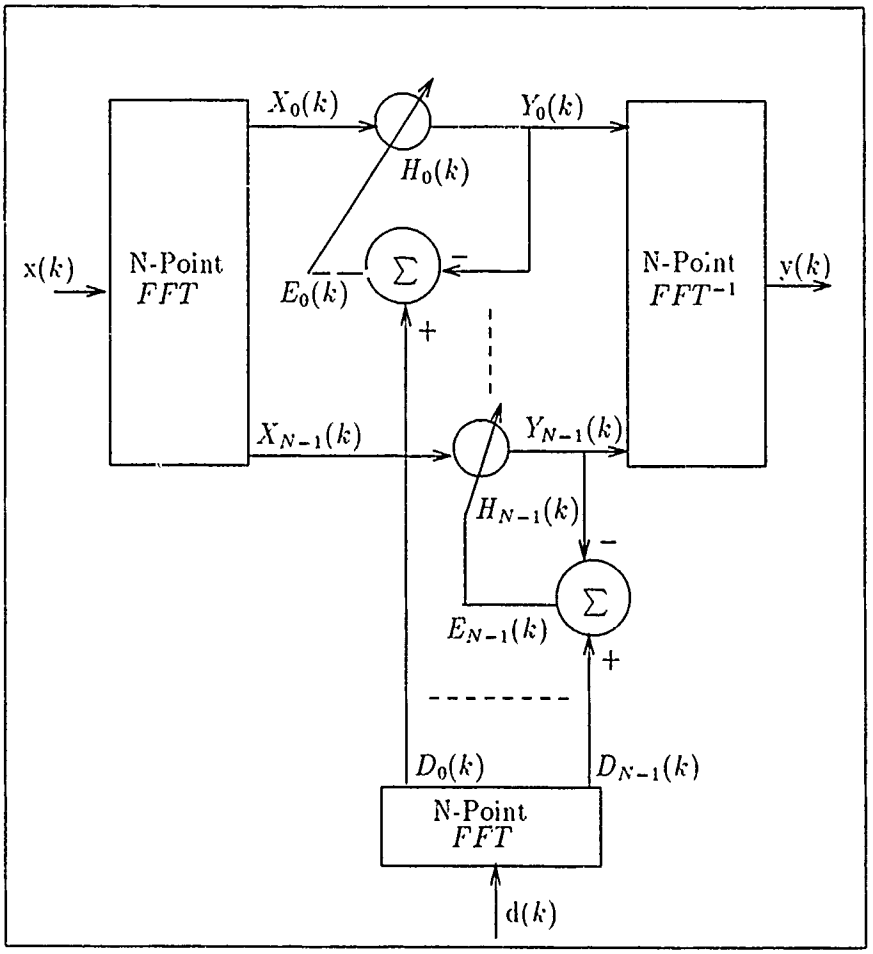

Figure 2.3. Frequency-domain Adaptive Filter

FDF1 Time-domain Input Vector Definition. The first step in presenting the frequency-domain equations is to define the filter input vectors. Letting x_i represent the input sequence, the N input sequence values which define the kth input block can be represented by x_{kN+i}, where $i = 0, 1, \ldots, N-1$. Using this notation, the N-point kth block input vector is defined

$$\mathrm{x}(k) = [x_{kN} \quad \ldots \quad x_{kN+N-1}]^T \tag{2.35}$$

Applying the same notation to the desired time-domain sequence, the associated kth block desired vector is represented as

$$\mathrm{d}(k) = [d_{kN} \quad \ldots \quad d_{kN+N-1}]^T \tag{2.36}$$

FDF1 Frequency-domain Input. The kth block input vector transform components define the diagonal components of the kth block input FFT matrix X(k) [2:148]:

$$\begin{aligned} \mathrm{X}(k) &= diag\{\mathcal{F}\{[x_{(kN)} \quad \ldots \quad x_{(kN+N-1)}]^T\}\} \\ &= \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{N-1}(k) \end{bmatrix} \end{aligned} \tag{2.37}$$

where $\mathcal{F}$ represents the forward FFT operator.

The kth block desired vector transform components define the kth block frequency-domain desired vector D(k), defined by

$$\mathrm{D}(k) = \mathcal{F}\{[d_{(kN)} \quad \ldots \quad d_{(kN+N-1)}]^T\} \tag{2.38}$$

and

$$\mathrm{D}(k) = \begin{bmatrix} D_0(k) \\ D_1(k) \\ \vdots \\ D_{N-1}(k) \end{bmatrix} \tag{2.39}$$

FDF1 Output Calculation. The transform components of the kth input block, $X_n(k)$, are multiplied with the respective kth block frequency-domain filter weights, $H_n(k)$, to produce the kth block output spectral components, $Y_n(k)$ [2:148-149]:

$$\mathrm{Y}(k) = \mathrm{X}(k)\mathrm{H}(k) \tag{2.40}$$

where

$$\mathrm{H}(k) = [H_0(k)\ \ H_1(k)\ \ \ldots\ \ H_{N-1}(k)]^T \tag{2.41}$$

Taking the inverse FFT of $\mathrm{Y}(k)$ produces the kth block output vector. Representing the N output sequence values as y_{kN+i}, where $i = 0, 1, \ldots . N-1$, the kth block time-domain output block vector is defined

$$\mathrm{y}(k) = [y_{(kN)}, \cdots, y_{(kN+N-1)}]^T \tag{2.42}$$

and

$$\mathrm{y}(k) = \mathcal{F}^{-1}\{\mathrm{X}(k)\mathrm{H}(k)\} \tag{2.43}$$

The $\mathcal{F}^{-1}$ operator used in Equation 2.43 is the inverse FFT operator.

FDF1 Frequency-domain Weight Update. Analogous to the time-domain LMS filter weight update, the output spectral components, $Y_n(k)$, are subtracted from the corresponding desired signal spectral components, $D_n(k)$, to produce the kth block frequency-domair error components, $E_n(k)$ [2:149]:

$$\mathrm{E}(k) = \mathrm{D}(k) - \mathrm{Y}(k) \tag{2.44}$$

The $E_n(k)$ values are used to update the frequency-domain filter weights, $H_n(k)$

$$\begin{aligned} \mathrm{H}(k+1) &= \mathrm{H}(k) + \mu \mathrm{X}^*(k)\mathrm{E}(k) \\ &= \mathrm{H}(k) + \mu[\mathrm{X}^*(k)\mathrm{D}(k) - \mathrm{X}^*(k)\mathrm{X}(k)\mathrm{H}(k)] \\ &= \mathrm{H}(k) + \mu \nabla_{F1}(k) \end{aligned} \tag{2.45}$$

where μ is again the convergence constant, $X^*(k)$ is the complex conjugate of $X(k)$, and $\nabla_{F1}(k)$ represents the frequency-domain gradient vector for FDF1 [2:149]. $\nabla_{F1}(k)$ is defined

$$\nabla_{F1}(k) = \begin{bmatrix} \nabla_{F1_0}(k) \\ \nabla_{F1_1}(k) \\ \vdots \\ \nabla_{F1_{N-1}}(k) \end{bmatrix} = \begin{bmatrix} X_0^*(k)[D_0(k) - X_0(k)H_0(k)] \\ X_1^*(k)[D_1(k) - X_1(k)H_1(k)] \\ \vdots \\ X_{N-1}^*(k)[D_{N-1}(k) - X_{N-1}(k)H_{N-1}(k)] \end{bmatrix} \tag{2.46}$$

Equation 2.45 shows that the FDF1 Weight update is accomplished once per block as opposed to each discrete time, i, increment.

For the previous discussion, in all cases, the index n indicates the spectral component, k indicates the block being processed, and i is the discrete time index. Also, because the FFT of a discrete time signal results in complex values, the frequency-domain components $H_n(k)$, $E_n(k)$, $X_n(k)$, $D_n(k)$, and $Y_n(k)$ are generally complex.

For each block of input data processed, the filter attempts to minimize the MSE between the desired spectral components and the input spectral components. This filter requires stationary inputs for the weights to converge.

FDF1 Time-domain Representation. The equivalent time-domain representation of Equation 2.45 is as follows [2:149]:

$$\mathrm{h}(k+1) = \mathrm{h}(k) + \mu[\boldsymbol{\chi}^T(k)\mathrm{d}(k) - \boldsymbol{\chi}^T\boldsymbol{\chi}(k)\mathrm{h}(k)] \tag{2.47}$$

where

$$\mathrm{h}(k) = \mathcal{F}^{-1}\mathrm{H}(k) \tag{2.48}$$

with the vector format specified as

$$\mathrm{h}(k) = [h_0(k) \;\; h_1(k) \;\; \ldots \;\; h_{N-1}(k)]^T \tag{2.49}$$

The symbol $\boldsymbol{\chi}(k)$ represents a circulant matrix [2:149] given by

$$\boldsymbol{\chi}(k) = \mathcal{F}^{-1}\mathrm{X}(k)\mathcal{F} \tag{2.50}$$

and

$$\chi(k) = \begin{bmatrix} x_0(k) & x_{(N-1)}(k) & \dots & x_1(k) \\ x_1(k) & x_0(k) & \dots & x_2(k) \\ \vdots & \vdots & & \vdots \\ x_{N-1}(k) & x_{N-2}(k) & \dots & x_0(k) \end{bmatrix} \tag{2.51}$$

The first column of $\chi(k)$ is the input vector x(k) since x(k) is the inverse FFT of the diagonal elements of X(k). By expressing the ith row of $\chi(k)$ as $\mathrm{x}_i^T(k)$, Equation 2.47 can be rewritten as

$$\mathrm{h}(k+1) = \mathrm{h}(k) + \mu \sum_{i=0}^{N-1} [d_i(k)\mathrm{x}_i(k) - y_i(k)\mathrm{x}_i(k)] \tag{2.52}$$

with $y_i(k)$ representing the ith component, where $i = 0 \ldots N-1$, of the kth block output vector (Equation 2.43). The output vector y(k) contains the N output values for the kth output block of the filter. FDF1 output values for the kth block are calculated in the time-domain by performing the circular convolution of h(k) and x(k). Using the circulant matrix, the kth block output vector is defined [2:149]

$$\mathrm{y}(k) = \chi(k)\mathrm{h}(k) \tag{2.53}$$

Substituting $e_i(k) = d_i(k) - y_i(k)$. where $d_i(k)$ represents the ith component of d(k) (Equation 2.36), into Equation 2.52 produces

$$\mathrm{h}(k+1) = \mathrm{h}(k) + \mu \sum_{i=0}^{N-1} e_i(k)\mathrm{x}_i(k) \tag{2.54}$$

where $\mathrm{x}_i(k)$ once again represents the ith row of $\chi(k)$. Equation 2.54 reveals a distinct departure from the standard LMS algorithm relating to the gradient estimate. In this case, even though weight update occurs only once per block. the gradient estimate is calculated as a recursive sum over the input block.

FDF1 Optimum Weight Vector. The first step in deriving an FDF1 optimal time-domain weight vector expression is the derivation of an equivalent frequency domain expression. The optimum frequency-domain weight vector minimizes [2:150]

$$= E[(\mathrm{D}(k) - \mathrm{Y}(k))^*(\mathrm{D}(k) - \mathrm{Y}(k)]$$

$$= E[\mathrm{D}^*(k)\mathrm{D}(k)] - \mathrm{R}_{xd}^*\mathrm{H} - \mathrm{H}^*\mathrm{R}_{xd} + \mathrm{H}^*\mathrm{R}_{xx}\mathrm{H} \tag{2.55}$$

where

$$\mathrm{R}_{xd} = E[\mathrm{X}^*(k)\mathrm{D}(k)] \tag{2.56}$$

and

$$\mathrm{R}_{xx} = E[\mathrm{X}^*(k)\mathrm{X}(k)] \tag{2.57}$$

Since d and x are stationary, R_{xx} is a diagonal matrix and the nth diagonal element is given by $E[X_n^*(k)X_n(k)]$. The nth element of R_{xd} is $E[X_n^*(k)D_n(k)]$ [2:150]. Taking the gradient of Equation 2.55 with respect to H, setting the result equal to the zero vector, and solving for H_{opt} produces [2:150].

$$\mathrm{H}_{opt} = \mathrm{R}_{xx}^{-1}\mathrm{R}_{xd} \tag{2.58}$$

The optimum time-domain weight vector is calculated by taking the inverse FFT of Equation 2.58 which produces the optimum time-domain weight vector for a circularly convolving filter [2:150]:

$$\mathrm{h}_{opt} = \mathrm{r}_{xx}^{-1}\mathrm{r}_{xd} \tag{2.59}$$

where

$$\mathrm{r}_{xx} = \mathcal{F}^{-1}\mathrm{R}_{xx}\mathcal{F} \tag{2.60}$$

and

$$\mathrm{r}_{xd} = \mathcal{F}^{-1}\mathrm{R}_{xd}\mathcal{F} \tag{2.61}$$

The matrix r_{xx} is a circulant matrix, because R_{xx} is a diagonal matrix [2:150]. Characteristically, the first row of r_{xx} contains lags zero through $N-1$ of the circular autocorrelation function of the input x [2:150]. Using the linear autocorrelation function $\phi(i)$, the circular autocorrelation function at lag i, $\phi_c(i)$, can be expressed as [2:150]

$$\phi_c(i) = \frac{N-i}{N}\phi(i) + \frac{i}{N}\phi(i-N) \tag{2.62}$$

Similarly, the circular cross-correlation of x and d generates terms that comprise the elements of the vector r_{xd} [2:150].

FDF1 Computational Requirements. A computational reduction is achieved with the FDF1 filter versus the time-domain adaptive filter in terms of multiplication opera-

N	FDF1 Real Multiplies / LMS Real Multiplies
4	1.375
8	0.875
16	0.531
32	0.313
64	0.180
256	0.056
1024	0.017

Table 2.1. FDF1 vs LMS Real Multiplies

tions required [2:147]. The FDF1 filter uses three N-point FFTs and $2N$ complex multiplies to calculate N-output points.

An N-point FFT can be accomplished using an $N/2$-point FFT and $N/2$ complex multiplies [2:147]. Computationally, a $N/2$-point FFT requires $(N/4)\log_2(N/2) - N/2$ complex multiplies so that a total of $(N/4)\log_2(N/2)$ complex multiplies are performed in each N-point FFT [2:148]. Adding $2N$ complex multiplies to the complex multiplies associated with three N-point FFTs gives a total of $(3N/4)\log_2(N/2) + 2N$ complex multiplies per N output points produced. An N-tap time-domain LMS filter requires $2N^2$ real multiplies to produce N output data points [2:147].

Assuming four real multiplies is equivalent to one complex multiply, the ratio of FDF1 real multiplies to LMS real multiplies is

$$\frac{FDF1RealMultiplies}{LMSRealMultiplies} = \frac{3/2\log_2(N/2) + 4}{N} \tag{2.63}$$

The computational savings is significant for large filters, as demonstrated in Table 2.1.

2.2.2.2 Fast LMS Filter In general, adaptive filters which perform linear convolution are more useful for digital filtering [2:152]. The Fast least-mean-square adaptive filter (FLMS) [2:152-157] described in this section performs strictly linear convolution; as opposed to the strictly circular convolution model discussed earlier. This filter will be referred to as Frequency-domain Filter 2 (FDF2) throughout the remainder of this thesis. The FDF2 filter, depicted in Figure 2.4, is a block-processing adaptive filter that produces N output data values during each weight update cycle. In producing N output values, the

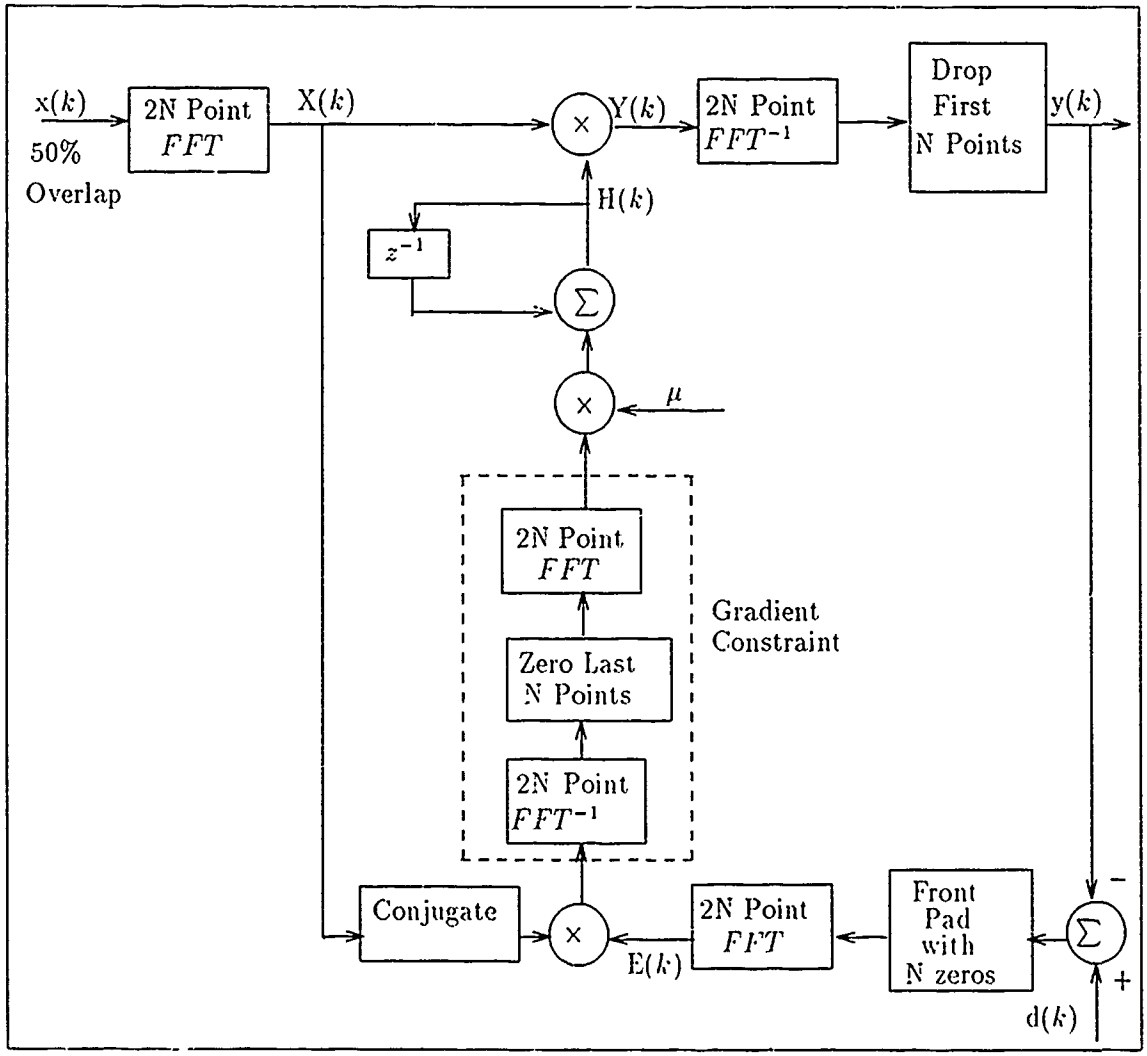

Figure 2.4. Frequency-domain Filter 2 (FDF2)

filter is utilizing $2N$ input data values, as opposed to the N values used by the FDF1 filter. The kth block input values consist of the N point current and N point previous input blocks.

FDF2 Time-domain Input Vector Definition. Using the notation x_i to represent the input sequence, the N input sequence values which define the kth input block can be represented by x_{kN+i}, where $i = 0, 1, \ldots, N-1$. Referring to Figure 2.4, the kth block input vector x(k) is composed of the concatenated N-point previous block and N-point current block which defines a 50% overlap of the $k-1$ and k N-point blocks. Therefore, the kth block input vector is defined

$$\mathrm{x}(k) = [\underbrace{x_{(kN-N)} \quad \cdots \quad x_{(kN-1)}}_{(k-1)th\ block} \quad \underbrace{x_{(kN)} \quad \cdots \quad x_{(kN+N-1)}}_{kth\ block}]^T \tag{2.64}$$

The kth block desired vector d(k) follows the FDF1 definition (Equation 2.36).

The additional input used during each block is due to the fact that the FLMS adaptive filter utilizes the "overlap-save" method in performing linear convolution. The "overlap-save" method "...corresponds to implementing an L-point circular convolution of a P-point impulse response h(k) with an L-point segment x(k) and identifying the part of the circular convolution that corresponds to a linear convolution. The resulting output segments are then patched together to form the output" [5:558]. Performing the circular convolution of an L-point sequence with a P-point sequence ($P < L$) results in an output sequence with the first ($P-1$) points incorrect in the context of linear convolution. The remaining points match the linear convolution result [5:558]. In this case, a 50% overlap is employed ($L/2 = N = P$) which has proven to be the most efficient [2:153].

FDF2 Frequency-domain Input. For the kth block, the $2N$ data points comprising the kth block input vector x(k) are transformed using the Fast Fourier Transform (FFT). The kth block input vector transform components define the diagonal components of the kth block input FFT matrix X(k) [2:153]:

$$\mathrm{X}(k) = diag\{\mathcal{F}[\underbrace{x_{(kN-N)} \quad \cdots \quad x_{(kN-1)}}_{(k-1)th\ block} \quad \underbrace{x_{(kN)} \quad \cdots \quad x_{(kN+N-1)}}_{kth\ block}]^T\}$$

$$= \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{2N-1}(k) \end{bmatrix} \tag{2.65}$$

where $\mathcal{F}$ is the forward FFT operator.

FDF2 Output Calculation. Representing the N output sequence values which define the kth output block as y_{kN+i}, where $i = 0, 1, \ldots, N-1$, the kth block time-domain output block vector is defined [2:153]

$$\begin{aligned} \mathrm{y}(k) &= [y_{(kN)}, \cdots, y_{(kN+N-1)}]^T \\ &= \textit{last } N \textit{ terms of } \mathcal{F}^{-1}\{\mathrm{X}(k)\mathrm{H}(k)\} \end{aligned} \tag{2.66}$$

where

$$\mathrm{H}(k) = [H_0(k) \;\; H_1(k) \;\; \ldots \;\; H_{2N-1}(k)]^T \tag{2.67}$$

FDF2 Frequency-domain Weight Update. After producing the kth output block the N time-domain error samples for the kth block are derived using

$$e_{kN+i} = d_{kN+i} - y_{kN+i} \tag{2.68}$$

where k represents the current block and $i = 0 \ldots N-1$. The time-domain error samples are transformed to create the frequency-domain error vector $\mathrm{E}(k)$, which is used to derive the $2N$-point FDF2 frequency domain gradient vector $\nabla_{F2}(k)$.

The FDF2 frequency-domain weight update algorithm differs from that used by FDF1 in that a gradient constraint is employed. The gradient constraint procedure is identified in Figure 2.4 as the dotted portion of the figure. The product of the $2N \times 2N$ kth block transform input matrix conjugate ($\mathrm{X}^*(k)$) and the $2N \times 1$ frequency-domain error vector ($\mathrm{E}(k)$) is the Gradient Constraint input. Since only N error terms are generated from the $2N$ input values used, the N-point kth time-domain error block sequence must be padded with N preceding zeros in order to generate a $2N$-point frequency domain error vector. Thus,

the frequency-domain error vector for the kth block is [2:154]:

$$\mathrm{E}(k) = \mathcal{F}\{[\underbrace{0 \ \ldots \ 0}_{N \text{ zeros}} \ \underbrace{(d_{(kN)} - y_{(kN)}) \ \ldots \ (d_{(kN+N-1)} - y_{(kN+N-1)})}_{k\text{th error block}}]^T\} \tag{2.69}$$

The first step in the Gradient Constraint procedure is to inverse transform the $\mathrm{X}^*(k)\mathrm{E}(k)$ product and save only the first N values. This result defines the FDF1 time-domain gradient vector [2:154]:

$$\nabla(k) = \textit{first } N \textit{ terms of } \mathcal{F}^{-1}\{\mathrm{X}^*(k)\mathrm{E}(k)\} \tag{2.70}$$

Next, the time-domain gradient vector (Equation 2.70) is zero end-padded with N zeros. Finally, the FDF2 frequency-domain gradient vector $\nabla_{F2}(k)$ is calculated by transforming the zero end-padded time-domain gradient vector.

Applying the definition for $\nabla_{F2}(k)$, the frequency-domain weight vector update is [2:154]

$$\begin{aligned} \mathrm{H}(k+1) &= \mathrm{H}(k) + \mu\mathcal{F}\begin{bmatrix} \nabla(k) \\ 0 \\ \vdots \\ 0 \end{bmatrix} \\ &= \mathrm{H}(k) + \mu\nabla_{F2}(k) \end{aligned} \tag{2.71}$$

where $\mathcal{F}$ is the forward FFT operator. The FDF2 frequency-domain gradient vector $\nabla_{F2}(k)$ qth term is

$$\begin{aligned} \nabla_{F2_q}(k) &= 1/2N \sum_{p=0}^{N-1} \sum_{r=0}^{2N-1} \sum_{i=N}^{2N-1} W_{2N}^{ri} W_{2N}^{-rp} W_{2N}^{qp} X_r^*(k) d_{(kN+i-N)} \\ &\quad -(1/2N)^2 \sum_{p=0}^{N-1} \sum_{r=0}^{2N-1} \sum_{i=N}^{2N-1} \sum_{n=0}^{2N-1} W_{2N}^{ri} W_{2N}^{-rp} W_{2N}^{-ni} W_{2N}^{qp} X_r^*(k) X_n(k) H_n(k) \\ q &= 0.1\ldots,2N-1 \end{aligned} \tag{2.72}$$

where

$$W_{2N} = e^{-j2\pi/2N} \tag{2.73}$$

The derivation of Equation 2.72 is a unique result of this thesis and is presented in Appendix E along with the $\nabla_{F2_q}(k)$ terms for the $N = 2$ case. Comparing Equations 2.46 and 2.72, the $\nabla_{F1_r}(k)$ terms contain only the $X_r^*(k)X_r(k)H_r(k)$ product, where $r = 0, 1, \ldots, N-1$. The $\nabla_{F2_q}(k)$ terms contain the $X_q^*(k)X_q(k)H_q(k)$ product as well as $X_i^*(k)X_q(k)H_q(k)$ products; where $i = 0, 1, \ldots, 2N-1$ and $i \neq q$. Thus, the $\nabla_{F2}(k)$ vector components utilize more filtering information than the $\nabla_{F1}(k)$ components. The additional information present in the FDF2 gradient is essentially an average of terms associated with each bin. Theoretically then, with a noisy input the FDF2 taps should converge with less adaptation noise and generally converge slower relative to the FDF1 taps.

Because the FFT of a discrete time signal generates complex values, the frequency-domain components of H(k), X(k), and E(k), are generally complex.

FDF2 Time-domain Representation. Having defined the FDF2 filter frequency domain operation, the equivalent time-domain representation can now be discussed. Applying the linear property of the DFT, the FDF2 time-domain weight-update equation is defined by the inverse FFT of Equation 2.71 [2:152]. This relationship requires the $k = 0$ frequency-domain weight vector, H(0), to be initialized to zero. The inverse FFT of Equation 2.71 provides

$$\mathrm{h}(k+1) = \mathrm{h}(k) + \mu \begin{bmatrix} \nabla(k) \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.74}$$

Thus, the gradient constraint and a requirement that the initial $2N$-tap weight vector H(k) be initialized to all zeros, produces a $2N$-tap time-domain weight vector with the last N values equal to zero. This result is due to the fact that a constant zero valued gradient is added to the last N time-domain taps on each weight update, which effectively results in an N term time-domain weight vector that can be specified as

$$\mathrm{h}(k) = [h_0(k) \;\; h_1(k) \;\; \ldots \;\; h_{N-1}(k)]^T \tag{2.75}$$

Therefore, the frequency-domain versus time-domain FDF2 weight vector relationship can be expressed as [2:153]

$$\mathrm{H}(k) = \mathcal{F}\{[\mathrm{h}^T(k) \;\; \underbrace{0 \;\; \ldots \;\; 0}_{N \text{ zeros}}]^T\} \tag{2.76}$$

The weight-update equation for the jth tap can be expressed as

$$h_j(k+1) = h_j(k) + \mu\nabla_j(k) \tag{2.77}$$

where $j = 0 \ldots N-1$. $\nabla_j(k)$ represents the jth component of $\nabla(k)$ which was defined in Equation 2.70. $\nabla_j(k)$ is defined [2:153]

$$\nabla_j(k) = \sum_{i=0}^{N-1} e_{(kN+i)} x_{(kN+i-j)} \tag{2.78}$$

where x_{kN+i} and e_{kN+i} represent the ith ($i = 0, \ldots, N-1$) component of the kth N-point block of the input and error sequences. Equation 2.78 shows the kth block gradient term for each time-domain weight to be the cross-correlation of the error terms derived from each block and the filter input [2:154].

With the time-domain weight vector defined, the FDF2 time-domain output calculation can be presented. Since the FDF2 filter processes blocks of data, the output calculation is most conveniently represented using vector multiplication. The linear convolution time-domain output calculation for the kth block is defined [2:153]

$$y_{(kN+i)} = \mathrm{h}^T(k)\mathrm{x}(kN+i) \quad i = 0, 1, \ldots, N-1 \quad ; k = 0, 1, 2, \ldots \tag{2.79}$$

where $y_{(kN+i)}$ represents the ith component of $\mathrm{y}(k)$ (Equation 2.66) and $\mathrm{h}(k)$ is specified in Equation 2.75. Letting the quantity $(kN+i)$ represent a discrete time index, the vector $\mathrm{x}^T(kN+i)$ containing the N most current input data values at time $(kN+i)$ is specified by

$$\mathrm{x}(kN+i) = [x_{(kN+i)} \;\; x_{(kN+i-1)} \;\; \cdots \;\; x_{(kN+i-N+1)}]^T \tag{2.80}$$

FDF2 Optimum Weight Vector. A derivation of the optimum time-domain weight vector for the FDF2 filter is identical to the LMS filter derivation (Section 2.2.1 Equations 2.32). This is based on the fact that the FDF2 filter performs linear convolution. Using the result presented in Equation 2.32 and the relationship stated in Equation 2.76, the frequency-domain optimum weight vector H_{opt} can be expressed as

$$\mathrm{H}_{opt} = \mathcal{F}\{[\mathrm{h}_{opt}^T \; \underbrace{0 \; \ldots \; 0}_{N \text{ zeros}}]^T\} \tag{2.81}$$

N	FDF2 Real Multiplies / LMS Real Multiplies
4	4.50
8	2.875
16	1.750
32	1.031
64	0.594
256	0.188
1024	0.057

Table 2.2. FDF2 vs LMS Real Multiplies

FDF2 Computational Requirements. A computational analysis reveals that the FDF2 filter requires fewer multiplies for large filters than the time-domain adaptive filter [2:154]. The FDF2 filter uses five $2N$-point FFTs and $4N$ complex multiplies to produce N output data points.

A $2N$-point FFT can be performed using an N-point FFT and N complex multiplies [2:154]. Since an N-point radix-2 FFT requires approximately $(N/2)\log_2(N) - $ complex multiplies [2:154], a total of $(5N/2)\log_2(N)$ complex multiplies are required for the five FFTs. Combining the FFT complex multiplies and $4N$ complex multiplies gives a total of $(5N/2)\log_2(N) + 4N$ complex multiplies per N output values produced.

The time-domain LMS filter requires $2N^2$ real multiplies to produce N output points [2:154]. Assuming four real multiplies is equivalent to one complex multiply yields the following ratio:

$$\frac{FDF2RealMultiplies}{LMSRealMultiplies} = \frac{5\log_2 N + 8}{N} \tag{2.82}$$

The computational savings is significant for large filters. Table 2.2 contains the ratio of FDF2 real multiplies to time-domain LMS real multiplies versus N. In comparison to Table 2.1, the ratios in Table 2.2 are clearly larger. This makes sense due to the fact that FDF2 requires $10N\log_2 N + 16N$ real multiplies per N output points. FDF1 conversely, requires $3N\log_2(N/2) + 8N$ real multiplies.

2.3 Chapter Summary

Several major points presented in this chapter are now summarized before proceeding to the next chapter. To begin, the major points presented concerning the DWT were:

1. The Discrete Walsh Transform (DWT) is a real valued transform that is implemented in the same general fashion that the Discrete Fourier Transform (DFT) is implemented (See Section 2.1.2).
2. The DWT does not have a time-shift property (See Section 2.1.3).
3. The inverse DWT of the product of two N-point DWTs produces dyadic convolution (See Section 2.1.4).
4. Dyadic convolution uses modulo-2 addition to determine the recursive time shift, whereas linear convolution uses arithmetic addition.
5. Dyadic correlation and convolution are identical since modulo-2 addition and subtraction are equivalent.
6. The DWT requires no multiplications whereas the DFT requires N^2 complex multiplies for an N-point transfrom.
7. The real DWT matrix and the complex DFT matrix have similar symmetry characteristics.
8. Analogous to the DFT, Fast Walsh Transforms (FWT) exist to exploit the DWT matrix symmetry.

The implementation similarity of the DWT and DFT warrants research in applying the DWT to existing frequency-domain block-processing filter designs. A circular convolution frequency-domain filter (FDF1) (See Section 2.2.2.1), and the Fast LMS frequency-domain filter (FDF2) (See Section 2.2.2.2) were examined. The next chapter presents two Walsh-domain filters that are developed from the FDF1 and FDF2 designs.

III. Walsh-Domain Filter Design

In Chapter II, a circular convolution frequency-domain filter (FDF1) and a linear convolution frequency-domain filter (FDF2) were presented. In this chapter, the following developments are presented:

1. Section 3.1 presents a Walsh-domain extension of the FDF1 filter design, Walsh-domain Filter 1 (WDF1). The filter performs dyadic convolution and requires fewer multiplication operations than FDF1.
2. Section 3.1 also presents a new algorithm, the Walsh Transform LMS algorithm 1 (WLMS1) which provides a weight update algorithm for the real valued WDF1 Walsh-domain tap weights.
3. Section 3.2 presents a new Walsh-domain filter design, Walsh-domain Filter 2 (WDF2), that is developed from the FDF2 filter. The filter uses a modified "overlap-save" method and requires fewer multiplication operations than FDF2.

3.1 Walsh-Domain Filter 1 (WDF1)

This section presents the WDF1 filter which was developed by extending the FDF1 design into the Walsh-domain. To aid the reader, a discussion parallel to the FDF1 presentation in Section 2.2.2.1 will be used.

Implementation of the FDF1 design with the DWT is a direct application of Equations 2.16 and 2.18 from Section 2.1.4. The design accommodates the use of the DWT directly (Figure 3.1) with minor algorithm changes associated with replacing a complex valued transform with a real valued transform.

3.1.1 WDF1 Time-domain Input Vector Definition. WDF1 filter input vectors are as specified for the FDF1 filter. Letting x_i represent the input sequence, the N input sequence values which define the kth input block are represented by x_{kN+i}, where $i = 0, 1, \ldots, N-1$ and $k = 0, 1, 2, \ldots$. Using this notation, the kth block input vector is defined

$$\mathrm{x}(k) = [x_{kN} \quad \ldots \quad x_{kN+N-1}]^T \tag{3.1}$$

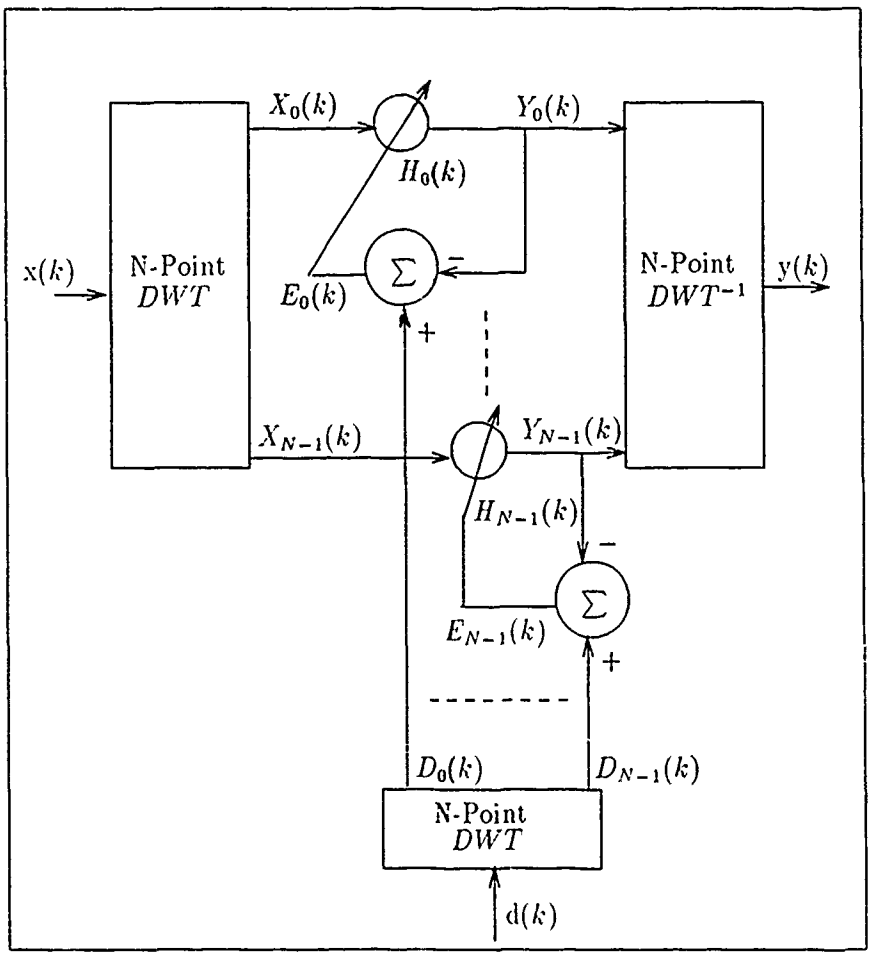

Figure 3.1. Walsh-domain Adaptive Filter 1 (WDF1)

Similarly, the kth block desired signal input vector is defined

$$\mathrm{d}(k) = [d_{kN} \quad \ldots \quad d_{kN+N-1}]^T \tag{3.2}$$

3.1.2 WDF1 Walsh-domain Input. Analogous to the FDF1 FFT input matrix (Equation 2.65, the kth block input DWT matrix $\mathrm{X}(k)$ is given by

$$\begin{aligned} \mathrm{X}(k) &= diag\{\mathcal{W}\{[x_{(kN)} \quad \ldots \quad x_{(kN+N-1)}]\}\} \\ &= \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{N-1}(k) \end{bmatrix} \end{aligned} \tag{3.3}$$

where $\mathcal{W}$ is the forward DWT operator. The kth block desired vector DWT components define the kth block Walsh-domain desired vector $\mathrm{D}(k)$:

$$\begin{aligned} \mathrm{D}(k) &= \mathcal{W}\{[d_{(kN)} \quad \ldots \quad d_{(kN+N-1)}]\} \\ &= \begin{bmatrix} D_0(k) \\ D_1(k) \\ \vdots \\ D_{N-1}(k) \end{bmatrix} \end{aligned} \tag{3.4}$$

3.1.3 WDF1 Output Calculation. The transform components of the kth input block, $X_n(k)$, are multiplied with the respective kth block Walsh-domain filter weights, $H_n(k)$, to produce the kth block output spectral components, $Y_n(k)$:

$$\mathrm{Y}(k) = \mathrm{X}(k)\mathrm{H}(k) \tag{3.5}$$

where

$$\mathrm{H}(k) = [H_0(k) \;\; H_1(k) \;\; \ldots \;\; H_{N-1}(k)]^T \tag{3.6}$$

Taking the inverse DWT of $\mathrm{Y}(k)$ produces the kth block output vector. Representing the N output sequence values as y_{kN+i}, where $i = 0, 1, \ldots, N-1$, the kth block time-domain

output block vector is defined

$$\mathrm{y}(k) = [y_{(kN)}, \ldots, y_{(kN+N-1)}]^T \tag{3.7}$$

and

$$\mathrm{y}(k) = \mathcal{W}^{-1}\{\mathrm{X}(k)\mathrm{H}(k)\} \tag{3.8}$$

The $\mathcal{W}^{-1}$ operator used in Equation 3.8 is the inverse DWT operator.

3.1.4 WDF1 Walsh-domain Weight Update. Following the same procedure used with the FDF1 filter, the output spectral components, $Y_n(k)$, are subtracted from the corresponding desired signal spectral components, $D_n(k)$, to produce the kth block Walsh-domain error components, $E_n(k)$:

$$\mathrm{E}(k) = \mathrm{D}(k) - \mathrm{Y}(k) \tag{3.9}$$

The $E_n(k)$ values are used to update the Walsh-domain filter weights.

Equation 2.45, which defines the FDF1 frequency-domain weight update algorithm, uses the conjugate of the input diagonal transform matrix $\mathrm{X}(k)$ in calculating the frequency-domain gradient. The corresponding Walsh-domain weight vector calculation is therefore

$$\mathrm{H}(k+1) = \mathrm{H}(k) + \mu\mathrm{X}(k)\mathrm{E}(k) \tag{3.10}$$

since the Walsh-domain components are real quantities. The vector format for $\mathrm{H}(k)$ is specified in Equation 3.6. Equation 3.10 defines Walsh Transform LMS algorithm 1 (WLMS1). A literature search indicates that WLMS1 has not previously been developed.

Using an approach similar to that used for Equation 2.45, Equation 3.10 can be expressed as

$$\begin{aligned} \mathrm{H}(k+1) &= \mathrm{H}(k) + \mu\mathrm{X}(k)\mathrm{E}(k) \\ &= \mathrm{H}(k) + \mu[\mathrm{X}(k)\mathrm{D}(k) - \mathrm{X}(k)\mathrm{X}(k)\mathrm{H}(k)] \\ &= \mathrm{H}(k) + \mu\nabla_{W1}(k) \end{aligned} \tag{3.11}$$

where μ is the convergence constant and $\nabla_{W1}(k)$ represents the Walsh-domain gradient vector for WDF1, i.e.

$$\nabla_{W1}(k) = \begin{bmatrix} \nabla_{W1_0}(k) \\ \nabla_{W1_1}(k) \\ \vdots \\ \nabla_{W1_{N-1}}(k) \end{bmatrix} = \begin{bmatrix} X_0(k)[D_0(k) - X_0(k)H_0(k)] \\ X_1(k)[D_1(k) - X_1(k)H_1(k)] \\ \vdots \\ X_{N-1}(k)[D_{N-1}(k) - X_{N-1}(k)H_{N-1}(k)] \end{bmatrix} \tag{3.12}$$

$\nabla_{F1}(k)$ and $\nabla_{W1}(k)$ are very similar (See Equations 2.46 and 3.12). The two vectors differ in that the $\nabla_{F1_j}(k)$ terms, where $j = 0, 1, \ldots, N-1$, are generally complex valued whereas the $\nabla_{W1_j}(k)$ terms are real valued.

For the previous discussion, in all cases, the index n indicates the spectral component, k indicates the block being processed, and i is the discrete time index. Because the DWT of a discrete time signal results in real values, the Walsh-domain components $H_n(k)$, $E_n(k)$, $X_n(k)$, $D_n(k)$, and $Y_n(k)$ are real.

3.1.5 WDF1 Time-domain Representation. In the preceding discussion it was demonstrated that replacement of the FFT with the DWT in the FDF1 design requires virtually no change to the transform domain equations presented for FDF1 (Section 2.2.2.1). The equivalent time-domain equations for the DWT implementation, however, present a departure from the FDF1 time-domain equations.

Using the relationship

$$\mathrm{h}(k) = \mathcal{W}^{-1}\{\mathrm{H}(k)\} \tag{3.13}$$

and the linear property of the DWT (Equation 2.7), the inverse DWT of Equation 3.10 gives

$$\mathrm{h}(k+1) = \mathrm{h}(k) + \mu\nabla(k) \tag{3.14}$$

where

$$\nabla(k) = \mathcal{W}^{-1}\{\mathrm{E}(k)\mathrm{X}(k)\} \tag{3.15}$$

Equations 2.18 and 2.15 state that the inverse DWT of the product of the two Walsh-domain vectors $\mathrm{E}(k)$ and $\mathrm{X}(k)$ is equivalent to the dyadic convolution of their time domain vectors. The dyadic convolution of the two time-domain vectors is analogous to the linear convolution

of these two vectors as described in Equation 2.78. The difference is the replacement of a modulo-2 shift for a linear addition derived shift and the incorporation of a scaling factor. These modifications provide

$$\nabla_j(k) = 1/N \sum_{i=0}^{N-1} e_i(k) x_{(i \oplus j)}, \quad j = 0, 1, \ldots, N-1 \tag{3.16}$$

where j is the time-domain weight index, $\oplus$ indicates modulo-2 addition for the binary representations of i and j, and $\nabla_j(k)$ defines the kth block gradient term for each time-domain weight. The $x_i(k)$ and $e_i(k)$ terms represent the ith component of x(k) and e(k) respectively, during the kth block:

$$\begin{aligned} \mathrm{x}(k) &= [x_{kN} \ \ldots \ x_{kN+N-1}] \\ &= [x_0(k) \ \ldots \ x_{(N-1)}(k)] \end{aligned} \tag{3.17}$$

and

$$\begin{aligned} \mathrm{e}(k) &= [e_{kN} \ \ldots \ e_{kN+N-1}] \\ &= [(d_{kN} - y_{kN}) \ \ldots \ (d_{kN+N-1} - y_{kN+N-1})] \\ &= [e_0(k) \ \ldots \ e_{(N-1)}(k)] \end{aligned} \tag{3.18}$$

Using Equation 3.16, the kth block weight update equation for the jth tap is defined

$$h_j(k+1) = h_j(k) + \mu \nabla_j(k) \tag{3.19}$$

Having defined the time-domain weight update equation, a similar equation for the output will fully specify WDF1 in the time-domain. Using the same approach applied in the derivation of Equation 3.16, Equation 3.8 can be expressed

$$y_i(k) = 1/N \sum_{i=0}^{N-1} h_j(k) x_{(i \oplus j)}, \quad i = 0, 1, \ldots, N-1 \tag{3.20}$$

where $y_i(k)$ is the ith component of the kth block output vector y(k).

Equation 3.20 states that y(k) is calculated by dyadically convolving the time-domain taps and input data values for the kth block. The time-domain output calculation can also

be represented in vector form. Letting the quantity $(kN+i)$ represent a discrete time index, where

$$\begin{aligned} i &= 0,1,\ldots,N-1 \\ k &= 0,1,\ldots \end{aligned} \tag{3.21}$$

the output sample at time $(kN+i)$ can be calculated using

$$y_{(kN+i)} = (1/N)\mathrm{h}^T(kN+i)\mathrm{x}(kN+i) \tag{3.22}$$

The vectors $\mathrm{x}(kN+i)$ and $h(kN+i)$ are defined

$$\begin{aligned} \mathrm{x}(kN+i) &= [x_{(kN+i)} \;\; x_{(kN+i\oplus 1)} \;\; \cdots \;\; x_{((kN+i)\oplus(N-1))}]^T \\ \mathrm{h}(kN+i) &= [h_0(kN+i) \;\; h_1(kN+i) \;\; \ldots \;\; h_{(N-1)}(kN+i)]^T \end{aligned} \tag{3.23}$$

where the terms of $\mathrm{x}(kN+i)$ are defined $x_{(kN+i\oplus j)}$, $j = 0,1,\ldots,N-1$, and $\oplus$ indicates modulo-2 addition for the binary representations of $(kN+i)$ and j.

3.1.6 WDF1 Optimum Weight Vector. The optimum time-domain weight vector for this design will differ from that presented in Section 2.2.1, due to the fact that the filter output calculations are performed using dyadic convolution and the autocorrelation matrix terms are calculated using dyadic correlation, which is mathematically equivalent to dyadic convolution (Section 2.1.5). The performance equation for WDF1 can be derived using vector representations for the output and error calculation. Having defined the kth block output vector calculation as a vector multiplication (Equation 3.20), it remains to define the error calculation in a similar manner. Using Equation 3.22 and the associated definitions, the error sample at time $(kN+i)$ is defined

$$\begin{aligned} e_{(kN+i)} &= d_{(kN+i)} - y_{(kN+i)} \\ &= d_{(kN+i)} - (1/N)\mathrm{h}^T(k)\mathrm{x}(kN+i) \end{aligned} \tag{3.24}$$

Making a change of variable such that the variable n is substituted for $(kN+i)$, Equation 3.22 becomes

$$y_n = (1/N)\mathrm{h}^T(n)\mathrm{x}(n) \tag{3.25}$$

where

$$\mathrm{x}(n) = [x_n \;\; x_{(n\oplus 1)} \;\; \ldots \;\; x_{(n\oplus(N-1))}]^T \tag{3.26}$$

and

$$\mathrm{h}(n) = [h_0(n) \;\; h_1(n) \;\; \ldots \;\; h_{(N-1)}(n)]^T \tag{3.27}$$

Similarly, Equation 3.24 becomes

$$\begin{aligned} \epsilon_n &= d_n - y_n \\ &= d_n - (1/N)\mathrm{h}^T(n)\mathrm{x}(n) \end{aligned} \tag{3.28}$$

Following the h_{opt} procedure outlined in Section 2.2.1; substitution of Equation 3.28 into Equation 2.26 produces

$$E[\epsilon_n^2] = E[d_n^2] + (1/N^2)\mathrm{h}^T R_w \mathrm{h} - (2/N)P_w^T \mathrm{h} \tag{3.29}$$

where R_w is the Walsh autocorrelation matrix and is defined as

$$R_w = E[\mathrm{x}(n)\mathrm{x}^T(n)] = \begin{bmatrix} \Phi_{xx}(0) & \Phi_{xx}(-1) & \ldots & \Phi_{xx}(-N+1) \\ \Phi_{xx}(1) & \Phi_{xx}(0) & \ldots & \Phi_{xx}(-N+2) \\ \Phi_{xx}(2) & \Phi_{xx}(1) & \ldots & \Phi_{xx}(-N+3) \\ \vdots & \vdots & \vdots & \vdots \\ \Phi_{xx}(N-1) & \Phi_{xx}(N-2) & \ldots & \Phi_{xx}(0) \end{bmatrix} \tag{3.30}$$

and $\Phi_{xx}(j) = E[x_n x_{(n\oplus j)}]$. P_w is the Walsh cross-correlation vector and is defined as

$$P_w = E[d_n \mathrm{x}(n)] = \begin{bmatrix} \Phi_{dx}(0) \\ \Phi_{dx}(-1) \\ \vdots \\ \Phi_{dx}(-N+1) \end{bmatrix} \tag{3.31}$$

where $\Phi_{dx}(j) = E[d_n x_{(n\oplus j)}]$. Under the assumption that the input signal and desired signal are stationary, R_w and P_w are constant and as such require no time subscript. Equation 3.29

is referred to as the performance equation or mean-square error (MSE) equation for the WDF1 filter.

Equation 3.29 characterizes a performance surface that the WDF1 filter searches to find a global minimum which is guaranteed to exist by the quadratic nature of the equation. The weight vector that satisfies the global minimum is referred to as h_{opt}. Solving for h_{opt} requires taking the gradient of $E[\epsilon_n^2]$ with respect to the h vector, and setting the gradient equal to the zero vector. Executing the same steps as indicated in Section 2.2.1, Equations 2.31 and 2.32, yields

$$h_{opt} = N R_w^{-1} P_w \tag{3.32}$$

Substituting Equation 3.32 into Equation 3.29 for h yields

$$\xi_{min} = E[d_n^2] - P_w^T R_w^{-1} P_w \tag{3.33}$$

Analogous to the time-domain optimum weight vector result (Equation 2.32), Equations 3.32 and 3.33 show that the WDF1 filter optimum weight vector is determined by the inverse of the input autocorrelation matrix and the cross-correlation vector. However, the autocorrelation and crosscorrelation terms for WDF1 are actually dyadic convolution terms (Section 2.1.5). Therefore, the dyadic autocorrelation input statistics and the dyadic cross-correlation statistics of the desired signal and input will directly affect the optimum weight vector.

3.1.7 WDF1 Computational Requirements. A computational comparison can be conducted for the WDF1 filter versus the FDF1 filter in the same manner as the LMS filter was compared to the FDF1 filter. The WDF1 filter uses three N-point DWTs to produce N output points. No multiplication operations are involved with the DWTs since an N-point DWT can be implemented with no multiplications and $N(N-1)$ adds. $2N$ real multiplies are required for weight update and output calculation. Therefore, the WDF1 filter requires $2N$ real multiplies per N output points versus the FDF1 requirement of $3N\log_2(N/2)+8N$. The ratio of WDF1 real multiplies to FDF1 real multiplies is

$$\frac{WDF1\,Real\,Multiplies}{FDF1\,Real\,Multiplies} = \frac{2}{3\log_2(N/2)+8} \tag{3.34}$$

Table 3.1 contains the ratio of WDF1 to FDF1 real multiplies for various N-point block sizes. Clearly, the WDF1 filter presents a computational savings in terms of multiplications.

N	WDF1 Real Multiplies / FDF1 Real Multiplies
4	0.180
8	0.143
16	0.118
32	0.100
64	0.087
256	0.069
1024	0.057

Table 3.1. WDF1 vs FDF1 Real Multiplies

3.2 Walsh-Domain Filter 2 (WDF2)

This section presents the WDF2 filter which was developed by extending the FDF2 design into the Walsh-domain. To aid the reader, a discussion parallel to the FDF2 presentation in Section 2.2.2.2 will be used.

Unlike the FDF1 filter, the FDF2 filter design is not suitable for use with the DWT. Figure 3.2 shows a block diagram of the final WDF2 configuration. A comparison with Figure 2.4 reveals a distinct difference in that the WDF2 filter does not use a Gradient Constraint. The Gradient Constraint was incompatible with the DWT and the reasons for that will be presented in Section 3.2.4.

3.2.1 WDF2 Time-domain Input Vector Definition. Like the FDF2 filter, WDF2 processes the current and previous input blocks to produce the current output block. To facilitate reader understanding, the notation presented for the FDF2 filter will again be presented at this time. Using the notation x_i to represent the input sequence, the N input sequence values which define the kth input block are represented by x_{kN+i}, where $i = 0, 1, \ldots, N-1$. Referring to Figure 3.2, the kth block input vector $\mathbf{x}(k)$ is composed of the concatenated N-point previous block and N-point current block, which defines a 50% overlap of the $k-1$

and k N-point blocks. Therefore, the kth block input vector is defined

$$\mathrm{x}(k) = [\underbrace{x_{(kN-N)} \ \cdots \ x_{(kN-1)}}_{(k-1)th\ block} \ \underbrace{x_{(kN)} \ \cdots \ x_{(kN+N-1)}}_{kth\ block}]^T \tag{3.35}$$

The kth block desired vector $\mathrm{d}(k)$ follows the WDF1 definition (Equation 3.2).

3.2.2 WDF2 Walsh-domain Input. The kth block DWT $2N \times 2N$ input matrix $\mathrm{X}(k)$ is specified using the DWT equivalent of Equation 2.65. In this case, the kth block input vector transform components define the diagonal components of the kth block input DWT matrix $\mathrm{X}(k)$:

$$\begin{aligned} \mathrm{X}(k) &= diag\{\mathcal{W}[\underbrace{x_{(kN-N)} \ \cdots \ x_{(kN-1)}}_{(k-1)th\ block} \ \underbrace{x_{(kN)} \ \cdots \ x_{(kN+N-1)}}_{kth\ block}]^T\} \\ &= \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{2N-1}(k) \end{bmatrix} \end{aligned} \tag{3.36}$$

where $\mathcal{W}$ is the forward DWT operator.

3.2.3 WDF2 Output Calculation. The kth block output vector ($\mathrm{y}(k)$) is determined using the DWT equivalent of Equation 2.66:

$$\begin{aligned} \mathrm{y}(k) &= [y_{(kN)}, \ldots, y_{(kN+N-1)}]^T \\ &= last\ N\ terms\ of\ \mathcal{W}^{-1}\{\mathrm{X}(k)\mathrm{H}(k)\} \end{aligned} \tag{3.37}$$

where

$$\mathrm{H}(k) = [H_0(k) \ \ H_1(k) \ \ \ldots \ \ H_{2N-1}(k)]^T \tag{3.38}$$

The FDF2 filter retains the last N values of the inverse transform result because they are equal to the values produced by the linear convolution sum (see Section 2.2.2.2). In Equation 3.37 the product $\mathrm{X}(k)\mathrm{H}(k)$ defines the dyadic convolution of the associated time-domain vectors for $\mathrm{X}(k)$ and $\mathrm{H}(k)$. Therefore, saving the last N, or any N-point combination of the $2N$ values produced, will not accomplish linear convolution. The choice of which output

samples to retain is then unclear. For this thesis, the last N values were chosen as the output, which is stated in Equation 3.37.

3.2.4 WDF2 Walsh-domain Weight Update. The FDF2 implementation was discussed in Section 2.2.2.2 and it was mentioned that the gradient constraint section of Figure 2.4 was necessary to ensure linear convolution. Section 2.1.4 presented the fact that the inverse Walsh Transform of the product of two N-point Walsh-transforms is equivalent to the dyadic convolution of their time-domain sequences (Equation 2.18). Therefore, the WDF2 filter will perform dyadic convolution whether the Gradient Constraint procedure is executed or not. As a result, using or not using the procedure fundamentally becomes a weight update issue.

The equations pertaining to the Gradient Constraint procedure from Section 2.2.2.2, replacing the FFT with DWT, are as follows:

$$\mathrm{E}(k) = \mathcal{W}\{[\underbrace{0 \quad \dots \quad 0}_{N \text{ zeros}} \quad \underbrace{(d_{(kN)} - y_{(kN)}) \quad \dots \quad (d_{(kN+N-1)} - y_{(kN+N-1)})}_{kth \text{ error block}}]^T\} \tag{3.39}$$

$$\nabla(k) = first\ N\ terms\ of\ \mathcal{W}^{-1}\{\mathrm{X}^*(k)\mathrm{E}(k)\} \tag{3.40}$$

$$\mathrm{H}(k+1) = \mathrm{H}(k) + \mu\mathcal{W}\begin{bmatrix} \nabla(k) \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.41}$$

where $\mathcal{W}$ represents the DWT operator.

In executing the last step of the Walsh-domain gradient calculation, as indicated by the vector transform in Equation 3.41, a zero end-padded vector is transformed using the DWT. Since the vector is end-padded with N zeros, the resulting $2N$ term Walsh domain gradient vector is component symmetric (Section 2.1.2.1). As a result of the component symmetric gradient vector, the $\mathrm{H}(k)$ vector will be component symmetric. Clearly then, the filter spectral bin taps are not going to converge independently. Therefore, the Gradient Constraint structure of Figure 2.1 is incompatible with a DWT based version of FDF2. Eliminating the 50% "over-lap save" method, which incorporates the Gradient Constraint, produces a filter equivalent to WDF1. However, eliminating the Gradient Constraint and

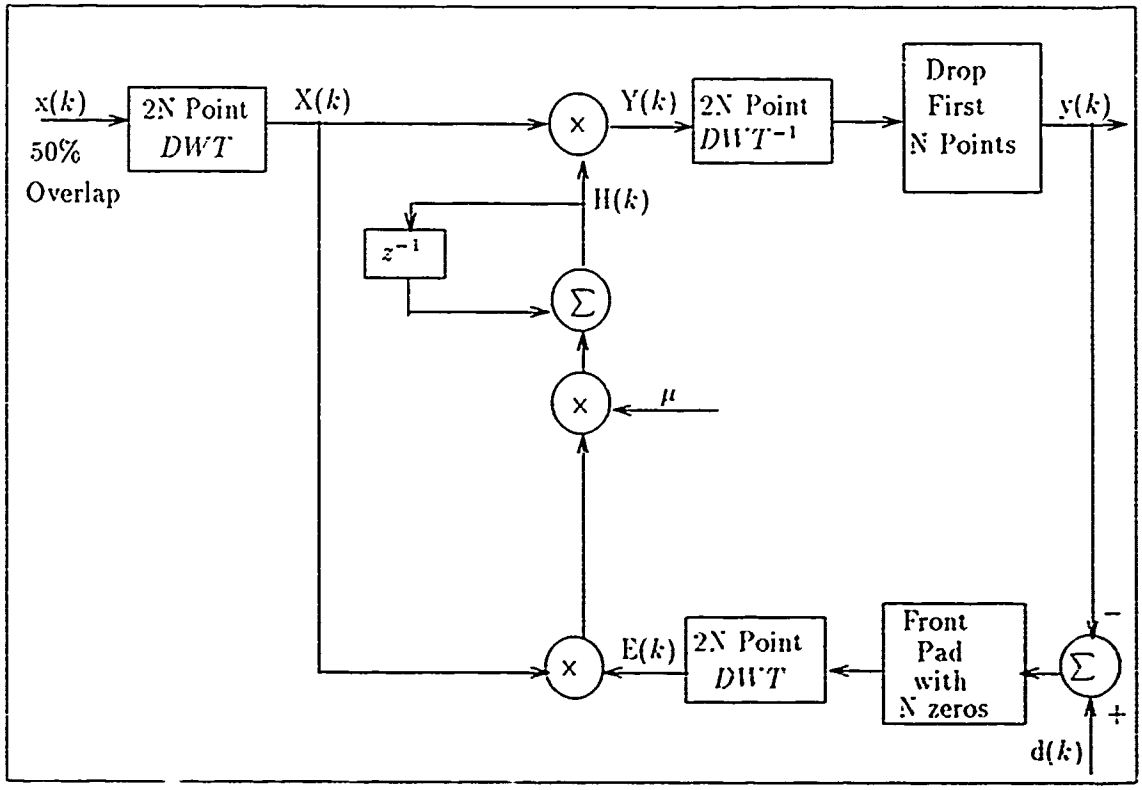

Figure 3.2. Walsh Adaptive Filter 2 (WDF2)

retaining the 50% over-lap characteristic presents a design unique with respect to the WDF1 filter.

Removing the Gradient Constraint block from the design leaves a Walsh-domain gradient vector that is identical in terms of calculation to that which appears in Equation 3.10:

$$H(k+1) = H(k) + \mu X(k)E(k) \tag{3.42}$$

where the $X(k)$, $E(k)$, and $H(k)$ vector components are real valued quantities. The product $X(k)E(k)$ defines the WDF2 Walsh-domain gradient vector $\nabla_{W2}(k)$. Comparatively, the WDF1 vector $\nabla_{W1}(k)$ and WDF2 vector $\nabla_{W2}(k)$ are unique since $E(k)$ for the WDF2 filter is derived from the zero front-padded kth block error vector (see Equation 3.39). Equation 3.42 will be referred to as Walsh Transform LMS algorithm 2 (WLMS2). A literature search indicates that WLMS2 has never been developed previously.

Using Equations 3.39, 3.37, and 3.36 the Walsh-domain WDF2 gradient vector for $N > 2$ is defined

$$\nabla_{W2}(k) = \begin{bmatrix} \nabla_{W2_0}(k) \\ \nabla_{W2_1}(k) \\ \nabla_{W2_2}(k) \\ \nabla_{W2_3}(k) \\ \vdots \\ \nabla_{W2_{2N-4}}(k) \\ \nabla_{W2_{2N-3}}(k) \\ \nabla_{W2_{2N-2}}(k) \\ \nabla_{W2_{2N-1}}(k) \end{bmatrix} \tag{3.43}$$

$$= 1/2 \begin{bmatrix} X_0(k)[D_{N_0}(k) - X_0(k)H_0(k) + X_1(k)H_1(k)] \\ X_1(k)[-D_{N_0}(k) + X_0(k)H_0(k) - X_1(k)H_1(k)] \\ X_2(k)[-D_{N_1}(k) - X_2(k)H_2(k) + X_3(k)H_3(k)] \\ X_3(k)[D_{N_1}(k) + X_2(k)H_2(k) - X_3(k)H_3(k)] \\ \vdots \\ X_{(2N-4)}(k)[D_{N_{N-2}}(k) - X_{(2N-4)}(k)H_{(2N-4)}(k) + X_{(2N-3)}(k)H_{(2N-3)}(k)] \\ X_{(2N-3)}(k)[-D_{N_{N-2}}(k) + X_{(2N-4)}(k)H_{(2N-4)}(k) - X_{(2N-3)}(k)H_{(2N-3)}(k)] \\ X_{(2N-2)}(k)[-D_{N_{N-1}}(k) - X_{(2N-2)}(k)H_{(2N-2)}(k) + X_{(2N-1)}(k)H_{(2N-1)}(k)] \\ X_{(2N-1)}(k)[D_{N_{N-1}}(k) + X_{(2N-2)}(k)H_{(2N-2)}(k) - X_{(2N-1)}(k)H_{(2N-1)}(k)] \end{bmatrix}$$

where the $D_{N_n}(k)$ terms represent the nth component of the N-point DWT of the current N-point block of the desired signal. For $N = 2$, $\nabla_{W2}(k)$ can be expressed as

$$\begin{bmatrix} \nabla_{W2_0}(k) \\ \nabla_{W2_1}(k) \\ \nabla_{W2_2}(k) \\ \nabla_{W2_3}(k) \end{bmatrix} = 1/4 \begin{bmatrix} X_0(k)[d_{(2k)} + d_{(2k+1)} - 2X_0(k)H_0(k) + 2X_1(k)H_1(k)] \\ X_1(k)[-d_{(2k)} - d_{(2k+1)} + 2X_0(k)H_0(k) - 2X_1(k)H_1(k)] \\ X_2(k)[-d_{(2k)} + d_{(2k+1)} - 2X_2(k)H_2(k) + 2X_3(k)H_3(k)] \\ X_3(k)[d_{(2k)} - d_{(2k+1)} + 2X_2(k)H_2(k) - 2X_3(k)H_3(k)] \end{bmatrix} \tag{3.44}$$

Equations 3.43 and 3.44 are derived in Appendix E.

Comparing Equations 3.12 and 3.43, the $\nabla_{W2_j}(k)$ terms contain $X_j(k)H_j(k)$ and $X_{(j+1)}(k)H_{(j+1)}(k)$ products, where $j = 0, 2, 4, \ldots, 2N-2$. Conversely, the $\nabla_{W1_j}(k)$ terms contain only the $X_j(k)H_j(k)$ product, where $j = 0, 1, \ldots, N-1$. The terms are distinct because a different E(k) is used for WDF2 (See Equations 3.39 and 2.44). This relationship is somewhat analogous to the associated FDF1 and FDF2 transform-domain gradient vector comparison.

The $\nabla_{W2_j}(k)$ components are distinctly different from the $\nabla_{F2_j}(k)$ components. The $\nabla_{F2_j}(k)$ components, aside from being generally complex valued, contain multiple product terms (See example in Appendix E) whereas the $\nabla_{W2_j}(k)$ components contain two product terms. Thus, the removal of the Gradient Constraint procedure reduces the averaging exhibited in the $\nabla_{F2}(k)$ vector components to two terms for the $\nabla_{W2}(k)$ components.

3.2.5 WDF2 Time-domain Representation. Having defined the Walsh-domain structures of WDF2, the time-domain representation of this filter can now be addressed. The time-domain weight-update algorithm is derived from the inverse DWT of Equation 3.42. Taking the inverse DWT of the product E(k)X(k) yields the time-domain dyadic convolution result of the two vectors

$$\mathrm{x}(k) = [\underbrace{x_{(kN-N)} \ \cdots \ x_{(kN-1)}}_{(k-1)th\ block} \ \underbrace{x_{(kN)} \ \cdots \ x_{(kN+N-1)}}_{kth\ block}] \} \tag{3.45}$$

$$\mathrm{e}(k) = [\underbrace{0 \ \ldots \ 0}_{N\ zeros} \ \underbrace{(d_{(kN)} - y_{(kN)}) \ \cdots \ (d_{(kN+N-1)} - y_{(kN+N-1)})}_{kth\ error\ block}]^T \tag{3.46}$$

The dyadic convolution result defines the time-domain WDF2 gradient vector $\nabla(k)$. The jth component of $\nabla(k)$ is defined

$$\nabla_j(k) = 1/2N \sum_{i=0}^{2N-1} e_i(k) x_{(i \oplus j)}(k), \quad j = 0, 1, \ldots, 2N-1 \tag{3.47}$$

where j is the time-domain weight index, $\oplus$ indicates modulo-2 addition for the binary representations of i and j, and $\nabla_j(k)$ defines the kth block gradient term for each time-domain weight. The $x_i(k)$ and $e_i(k)$ terms represent the ith component of x(k) and e(k)

respectively, during the kth block:

$$\begin{aligned} \mathrm{x}(k) &= [\underbrace{x_{(kN-N)} \ \cdots \ x_{(kN-1)}}_{(k-1)th\ block} \ \underbrace{x_{(kN)} \ \cdots \ x_{(kN+N-1)}}_{kth\ block}] \\ &= [x_0(k) \ \cdots \ x_{(2N-1)}(k)] \end{aligned} \tag{3.48}$$

and

$$\begin{aligned} \mathrm{e}(k) &= [\underbrace{0 \ \cdots \ 0}_{N\ zeros} \ \underbrace{(d_{(kN)} - y_{(kN)}) \ \cdots \ (d_{(kN+N-1)} - y_{(kN+N-1)})}_{kth\ error\ block}]^T \\ &= [e_0(k) \ \cdots \ e_{(2N-1)}(k)]^T \end{aligned} \tag{3.49}$$

Using Equation 3.47, the time-domain weight update equation can be specified

$$h_j(k+1) = h_j(k) + \mu \nabla_j(k) \tag{3.50}$$

The time-domain weight index j now spans $2N$ taps as opposed to the FDF2 case of N taps, because the gradient constraint was eliminated. For a block size of N input data values, the WDF2 filter has a time-domain equivalent filter with $2N$ taps as opposed to the FDF2 filter which has N time-domain equivalent taps. The number of output values remains N.

The time-domain equivalent of Equation 3.37 is determined using the relationships stated in Equations 2.18 and 2.15 from Section 2.1.4. Equation 2.18 states that the inverse DWT of the product $\mathrm{X}(k)\mathrm{H}(k)$ yields the dyadic convolution of the associated time-domain vectors $\mathrm{x}(k)$ and $\mathrm{h}(k)$. The dyadic convolution sum for the $2N$-point kth block input vector $\mathrm{x}(k)$ and time-domain tap vector $\mathrm{h}(k)$ is given by

$$y_i(k) = 1/2N \sum_{j=0}^{2N-1} h_j(k) x_{(i \oplus j)}(k), \quad i = 0, 1, \ldots, 2N-1 \tag{3.51}$$

where $x_i(k)$ is the ith component of $\mathrm{x}(k)$ (see Equation 3.48), and $\oplus$ indicates modulo-2 addition for the binary representations of i and j. The last N samples of this result represent the values $y_{(kN+j)}$, where $j = 0, \ldots, N-1$, and k is the block index. Therefore,

the WDF2 kth block output can be expressed as

$$y_i(k) = 1/2N \sum_{j=0}^{2N-1} h_j(k) x_{((i+N)\oplus j)}(k), \quad i = 0, 1, \ldots, N-1 \tag{3.52}$$

where $y_i(k)$ represents the ith component of $\mathrm{y}(k)$ such that

$$\begin{aligned} \mathrm{y}(k) &= [y_{(kN)} \quad \cdots \quad y_{(kN+N-1)}]^T \\ &= [y_0(k) \quad \cdots \quad y_{(N-1)}(k)]^T \end{aligned} \tag{3.53}$$

The output calculation in vector form can be expressed as

$$y_{(kN+i)} = \mathrm{h}^T(kN+i)\mathrm{x}(kN-N+i) \quad j = 0, 1, \ldots, N-1 \tag{3.54}$$

where

$$\mathrm{x}(kN-N+i) = [x_{(kN-N)} \;\; x_{(kN-N+1)} \;\; \cdots \;\; x_{(kN+N-1)}]^T \tag{3.55}$$

and $\mathrm{h}(k)$ is specified as

$$\mathrm{h}(kN+i) = [h_{(i+N)}(k) \;\; h_{((i+N)\oplus 1)}(k) \;\; \cdots \;\; h_{((i+N)\oplus(2N-1)}(k)]^T \tag{3.56}$$

In comparison, both WDF1 and WDF2 kth block output vectors are determined from the dyadic convolution of $\mathrm{x}(k)$ and $\mathrm{h}(k)$. The distinction between WDF2 and WDF1 output calculation, aside from input vector and tap vector length, is that only the last N values of the WDF2 dyadic convolution result are saved as the N output values for the kth block whereas the N-point WDF1 dyadic convolution result represents that filters kth block output. In both cases the time-domain weight vector is the same size as the input vector.

3.2.6 WDF2 Optimum Weight Vector. Given that the output values are derived from the dyadic convolution of the input and time-domain weight vector in both cases, the WDF2 MSE equation is the same as that produced for WDF1 (Equation 3.29) with the constant term changed from N to $2N$ to account for the fact that the WDF2 filter is convolving $2N$ size vectors.

Replacing the N term in Equation 3.29 with $2N$, yields the following WDF2 MSE equation

$$E[\epsilon_n^2] = E[d_n^2] + (1/4N^2)\mathrm{h}^T R_w \mathrm{h} - (1/N)P_w^T \mathrm{h} \tag{3.57}$$

where the matrix R_w and the vector P_w are as specified in Section 3.1, Equations 3.30 and 3.31. Applying the optimum weight vector derivation procedure outlined in Section 2.2.1, the optimum time-domain weight vector for WDF2 is

$$\mathrm{h}_{opt} = 2N R_w^{-1} P_w \tag{3.58}$$

Substituting Equation 3.58 into Equation 3.57 for h yields

$$\xi_{min} = E[d_n^2] - P_w^T R_w^{-1} P_w \tag{3.59}$$

which is the same result that was derived for WDF1 (Equation 3.33). Theoretically this is a reasonable result given that the optimum time-domain weight vectors differ only by a factor of 2. Equations 3.58 and 3.59 show that the WDF2 filter optimum weight vector is determined by the inverse of the input dyadic autocorrelation matrix and the dyadic cross-correlation vector. Therefore, as in the case of the WDF1 filter, the dyadic autocorrelation input statistics and the dyadic cross-correlation statistics of the desired signal and input determine the optimum weight vector.

3.2.7 WDF2 Computational Requirements. A computational comparison can be conducted for the WDF2 filter versus the FDF2 filter in the same manner as the LMS filter versus the FDF2 filter. The WDF2 filter uses three $2N$-point DWTs to produce N output points each requiring no multiplications since a $2N$-point DWT can simply be implemented with $2N(2N-1)$ adds. $4N$ real multiplies are required for weight update and output calculation. Therefore, the WDF2 filter requires $4N$ real multiplies per N output points versus the FDF2 requirement of $10N\log_2(N) + 16N$. The ratio of WDF2 real multiplies to FDF2 real multiplies is

$$\frac{WDF2RealMultiplies}{FDF2RealMultiplies} = \frac{2}{5\log_2(N) + 8} \tag{3.60}$$

Table 3.2 summarizes the computational savings related to multiplications.

N	WDF2 Real Multiplies / FDF2 Real Multiplies
4	0.111
8	0.086
16	0.071
32	0.061
64	0.053
256	0.042
1024	0.034

Table 3.2. WDF2 vs FDF2 Real Multiplies

3.3 Chapter Summary

Two adaptive Walsh-domain filters were developed by extending the FDF1 and FDF2 filter designs into the Walsh-domain. The important developments presented are as follows:

1. The FDF1 filter supports using the DWT with no modification to the design. This implementation was termed Walsh-domain filter 1 (WDF1) (See Section 3.1).
2. The gradient constraint portion of the FDF2 design is incompatible with the DWT. Removal of the gradient constraint does result in a DWT compatible design. This implementation was termed Walsh-domain Filter 2 (WDF2) (See Section 3.2).
3. The WDF1 and WDF2 Walsh-domain weight update equations (WLMS1 and WLMS2) are based on the complex LMS algorithm and they are very similar but distinct.
4. The dyadic autocorrelation input statistics and the dyadic cross-correlation and autocorrelation statistics of the desired signal and input determine the optimum weight vector for WDF1 and WDF2.
5. A Walsh-domain LMS algorithm was developed for the first time in this thesis.

The next chapter discusses the implementation and verification of WDF1 and WDF2 software implementations.

IV. Filter Verification

4.1 Introduction

The previous chapter presented a theoretical analysis of two Walsh-domain adaptive filters; designated WDF1 and WDF2. The next step is to implement the filters in software and then verify the software implementations. Therefore, there are three major goals for this chapter which are presented in the remainder of this section.

1. The equations and algorithms implemented in the WDF1 software are defined (Sections 4.2.3 and 4.2.2).
2. The equations and algorithms implemented in the WDF2 software are defined (Section 4.2.4 and 4.2.2).
3. The WDF1 and WDF2 filters are tested for errors (Sections 4.2.3.2, 4.2.4.2, and 4.3).

4.2 Software Algorithm Identification And Testing.

This section is intended to familiarize the reader with the WDF1 and WDF2 software and associated algorithms. This section also discusses the specific test used to verify each filter and the results of the test. A brief discussion concerning two filters used in this thesis for comparison purposes is also presented.

4.2.1 Comparison Filters. This thesis utilizes a time-domain LMS adaptive filter and a frequency-domain block-processing filter in Chapter V to evaluate relative performance measures for the WDF1 and WDF2 filters. The time-domain LMS adaptive filter will be referred to as the TDF filter throughout the remainder of this thesis. The TDF filter is a single-input adaptive transversal filter (Section 2.2.1) with a variable filter size. The FDF2 filter (Section 3.2) is used as a baseline for comparison. This filter uses a radix-2 FFT. Throughout the remainder of this thesis, this filter will be referred to as the FDF filter.

4.2.2 Gain Constant Calculation. There are several methods for calculating a gain constant μ based on the filter processing domain. Gain constant for the TDF filter is defined as

$$\mu = \frac{Misadjustment}{N(SignalEnergy)} = \frac{M}{N \cdot E[x_i^2]} \tag{4.1}$$

where N represents the number of filter taps and x_i is a stationary signal [3:3-6]. *Misadjustment* is defined as "...a measure of how closely the adaptive process tracks the true Weiner solution ..." [3:3-6]. A larger M value results in a larger μ value and produces faster adaptation but at a cost of greater adaptation noise.

There are two methods of calculating the gain constant, μ, for the block processing filters. The first uses the average power in each spectral bin. For the WDF1 and WDF2 filters, the nth bin gain constant, μ_n, is calculated as follows

$$\mu_n = \frac{Misadjustment}{E[P_{bin_n}(k)]} \tag{4.2}$$

where $P_{bin_n}(k) = X_n^2(k)$, k represents the kth block, and $X_n(k)$ represents the nth diagonal component of the input transform matrix $X(k)$. In the case of the FDF filter, the spectral bins and their respective filter weights are complex. Therefore, the FDF filter utilizes an independant μ for the real and imaginary parts of the bin. Calculation of μ_{real} and μ_{imag}, the real and imaginary gain constants respectively, using Equation 4.2 is accomplished by replacing $P_{bin_n}(k)$ with $P_{binreal_n}(k)$ and $P_{binimag_n}(k)$ respectively.

The second calculates a constant μ common to all of the spectral bins. This method calculates the average power in each bin with respect to the block index k, P_{bin_n}, and averages the P_{bin_n} values to produce P_{binavg}. For the WDF1 and WDF2 filters, the gain constant is calculated as follows

$$\mu = \frac{Misadjustment}{P_{binavg}} \tag{4.3}$$

where

$$P_{binavg} = 1/\eta \sum_{i=0}^{\eta-1} P_{bin_i} \tag{4.4}$$

and η represents the number of spectral bins (2N for WDF2 and N for WDF1). In the case of FDF2, the average of the sum of all the real and imaginary bin component powers is used to calculate P_{binavg}

$$P_{binavg} = 1/4N \sum_{i=0}^{\eta-1} (P_{binreal_i} + P_{binimag_i}) \tag{4.5}$$

where the $1/4N$ constant accounts for the 2N real bins and 2N imaginary bins.

4.2.3 WDF1 Filter The WDF1 program implements the equations developed in the previous chapter. In performing the DWT and inverse DWT, the program implements the direct form of Equations 2.3, 2.1, and 2.4.

4.2.3.1 WDF1 Software Overview The WDF1 software flow is as follows:

- Initialize variables, arrays, and vectors
- Load input signal and desired signal
- Calculate the number of N-point blocks in the input
- Calculate μ
- *Loop Start*: Create the N-point current block input vector
- Calculate the Walsh-domain N by N diagonal input matrix X(k)
- Calculate the Walsh-domain N-point desired block input vector D(k)
- Calculate the N-point output block vector y(k)
- Calculate the Walsh-domain N-point error vector E(k)
- Calculate the Walsh-domain N-point Gradient vector
- Update the N-point Walsh-domain weight vector using WLMS1
- Loop to *Loop Start* if more blocks

4.2.3.2 WDF1 Filter Verification Test This test verifies the WDF1 filter which includes Walsh Transform LMS algorithm 1 (WLMS1). The test input signal used is derived from summing the columns of the Walsh 8 by 8 (W_8) transform matrix (See Appendix A), and periodically repeating the resultant sequence. Summing the columns of W_8 produces an 8-point sequence of

$$x_k = \{8\ 0\ 0\ 0\ 0\ 0\ 0\ 0\} \tag{4.6}$$

with a corresponding DWT of

$$X_n = \{1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\} \tag{4.7}$$

The sequence expressed in Equation 4.6, with the filter configured to process 8-point blocks, is the input vector x(k) for the kth block processed. The desired input signal is produced

by first multiplying each row of W_8 by a different constant. The resulting modified W_8 matrix columns are then summed, producing an 8-point sequence, which is then repeated periodically. Utilizing the coefficients listed in Table 4.1 produces the 8-point sequence

$$d_k = \{18.15 \quad -6.85 \quad -2.934 \quad -0.984 \quad 0.411 \quad -5.607 \quad 1.033 \quad 6.652\} \tag{4.8}$$

with a corresponding DWT of

$$D_n = \{1.2341 \quad 0.6116 \quad 3.5123 \quad 0.2921 \quad 4.8234 \quad 1.9142 \quad 2.8314 \quad 2.9311\} \tag{4.9}$$

The sequence expressed in Equation 4.8, with the filter configured to process 8-point blocks, is the desired vector d(k) for the kth block processed. Since the input spectral components for each block processed all have a magnitude of 1 (Equation 4.7) and the corresponding desired block spectral components (Equation 4.9) are constant from block to block, the corresponding tap values for each bin should theoretically converge to the desired spectral component values. The input and desired signal are depicted in Figure 4.3.

Configuring the WDF1 filter as specified in Table 4.2, the WDF1 output error is depicted in Figure 4.4. Figures 4.9 thru 4.16 depict the adaptation tracks for the bin taps. Clearly, the figures show that WLMS1 is a valid Walsh-domain weight-update equation. All of the data in the figures and tables presented for this test were produced using a contant μ value for the bin taps. The results using an independantly derived μ value for each bin tap were identical and the figures and tables presented for this test are representative of those results. Table 4.3 displays the theoretical and experimental tap values for the WDF1 filter.

Row	Coefficient
0	1.234
1	0.611
2	3.512
3	0.292
4	4.823
5	1.914
6	2.831
7	2.931

Table 4.1. Verification Test coefficients

Parameter	*Setting*
Block Size	8
Misadjustment	0.2
Datasize	1000

Table 4.2. Verification Test WDF1 filter settings

Tap	*Theoretical*	*Experimental*
0	1.234	1.234
1	0.611	0.611
2	3.512	3.512
3	0.292	0.292
4	4.823	4.823
5	1.914	1.914
6	2.831	2.831
7	2.931	2.931

Table 4.3. WDF1 Verification Test Experimental Results

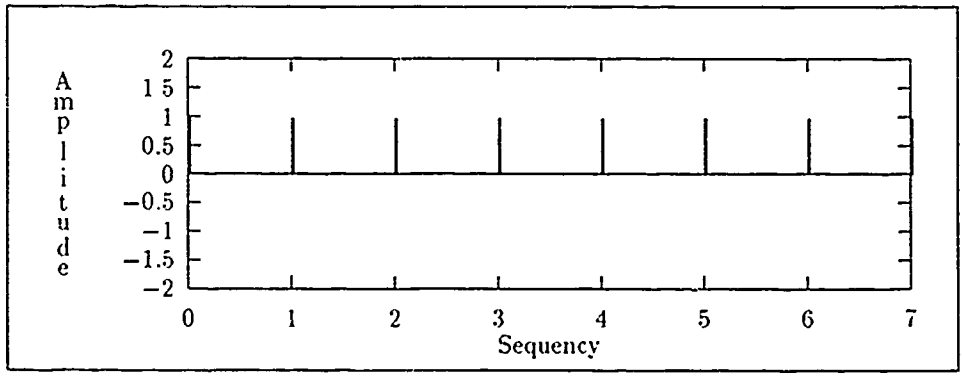

Figure 4.1. The DWT of one period of the Input Signal

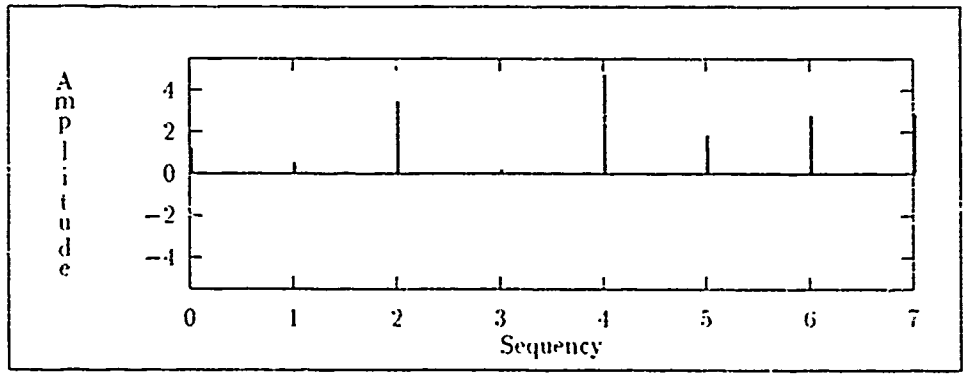

Figure 4.2. The DWT of one period of the Desired Signal

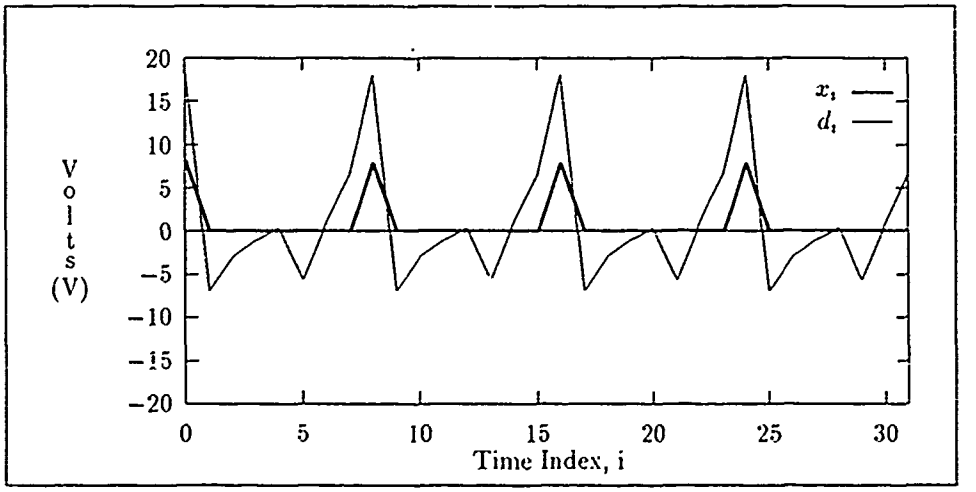

Figure 4.3. WDF1 Verification Test Input Signals

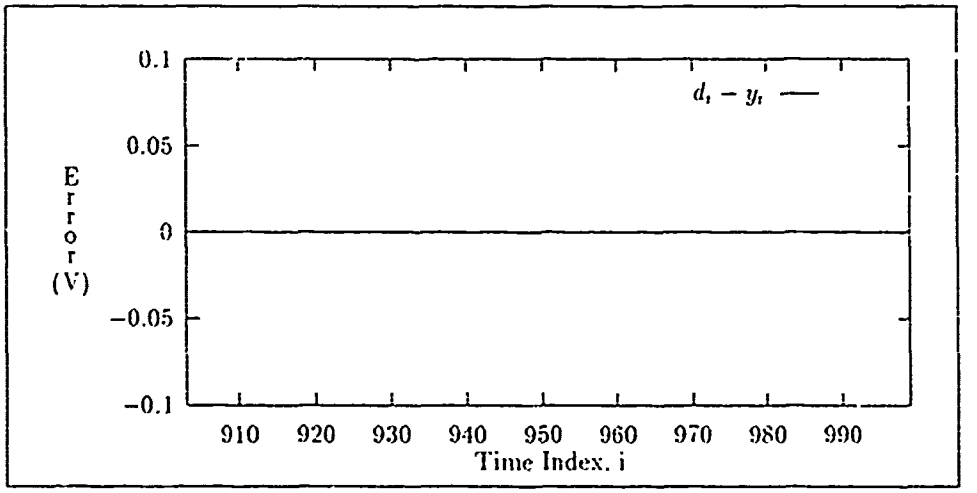

Figure 4.4. WDF1 Filter Verification Test Output Error.

4.2.4 WDF2 Filter The WDF2 program implements the equations developed in Section 3.2 of the previous chapter. As with the WDF1 program, the DWT and inverse DWT transforms are performed using Equations 2.3, 2.1, and 2.4.

4.2.4.1 WDF2 Software Overview The WDF2 software flow is as follows:

- Initialize variables, arrays, and vectors
- Load input signal and desired signal
- Calculate the number of N-point blocks in the input
- Calculate μ
- *Loop Start*: Create the $2N$-point past-current block input vector
- Calculate the Walsh-domain $2N$ by $2N$ diagonal input matrix $X(k)$
- Calculate the N-point output block vector $y(k)$
- Calculate the Walsh-domain $2N$-point error vector $E(k)$
- Calculate the Walsh-domain $2N$-point Gradient vector
- Update the $2N$-point Walsh-domain weight vector using WLMS2
- Loop to *Loop Start* if more blocks

4.2.4.2 WDF2 Filter Verification This test verifies the WDF2 filter software which includes Walsh Transform LMS algorithm 2 (WLMS2). The signal set used to verify the WDF1 filter is also used to verify the WDF2 filter. Since the WDF2 filter utilizes the concatenated current and previous input data blocks to evaluate the current block N output values, the input vector, excluding the first block processed, is given by

$$x(k) = \{8\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 8\ 0\ 0\ 0\ 0\ 0\ 0\ 0\} \tag{4.10}$$

with a corresponding DWT of

$$X_n = \{1.234\ 0\ 0\ 0.612\ 3.512\ 0\ 0\ 0.292\ 4.823\ 0\ 0\ 1.914\ 2.831\ 0\ 0\ 2.931\} \tag{4.11}$$

The $2N$-point DWT, with $N = 8$, of the desired sequence specified by Equation 4.8 for each block is

$$\begin{aligned} d(k) = \{&18.15 \ -6.85 \ -2.934 \ -0.984 \ 0.411 \ -5.607 \ 1.033 \ 6.652 \\ &18.15 \ -6.85 \ -2.934 \ -0.984 \ 0.411 \ -5.607 \ 1.033 \ 6.652\} \end{aligned} \quad (4.12)$$

with a corresponding DWT of

$$D_n = \{1.234 \ 0 \ 0 \ 0.612 \ 3.512 \ 0 \ 0 \ 0.292 \ 4.823 \ 0 \ 0 \ 1.914 \ 2.831 \ 0 \ 0 \ 2.931\} \quad (4.13)$$

The input vector and desired vector Walsh spectrums are illustrated in Figure 4.5 and 4.6. The figures show that the zero valued components are identical in number and occur in the same index positions. The nonzero valued components for each input block are of magnitude 1 and therefore the corresponding taps for each bin should theoretically converge to the desired bin spectral magnitudes.

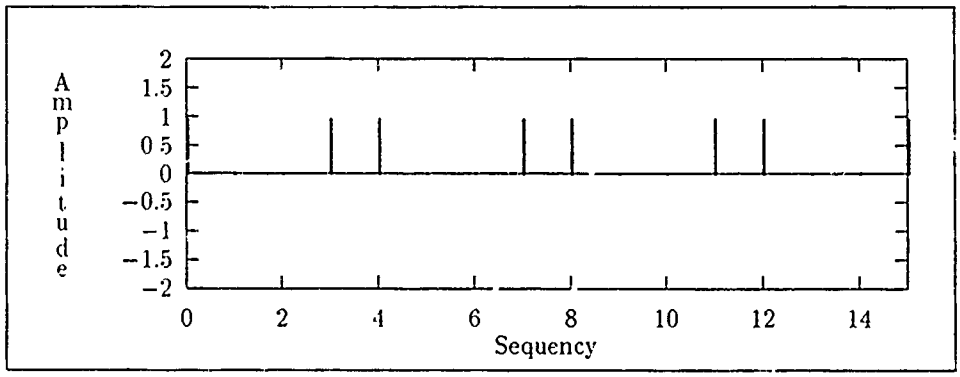

Figure 4.5. The 16-point DWT of two periods of the Input Signal

Configuring the WDF2 filter as specified in Table 4.2, the WDF2 output error is depicted in Figure 4.8. Table 4.4 displays the theoretical and experimental tap values for the WDF2 filter taps. The results displayed in the figures and tables were produced using a constant μ value for the bin taps. The previous input block for the initial current input block was initialized to zero.

Figure 4.7 depicts the DWT of the initial input vector processed by the filter. As Figure 4.7 shows, the initial Walsh-domain input spectrum is such that there are no zero

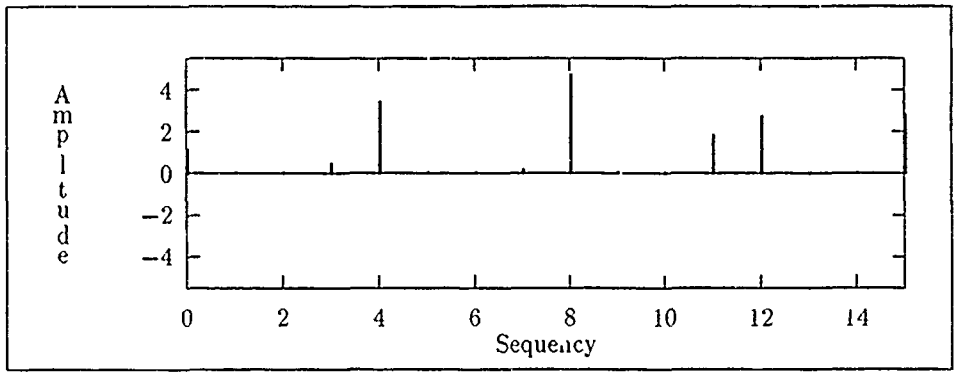

Figure 4.6. The 16-point DWT of two periods of the Desired Signal

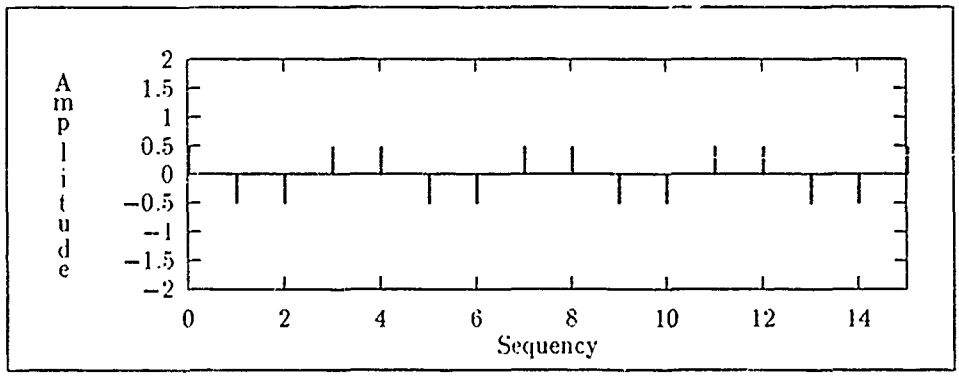

Figure 4.7. The 16-point DWT of the initial input vector

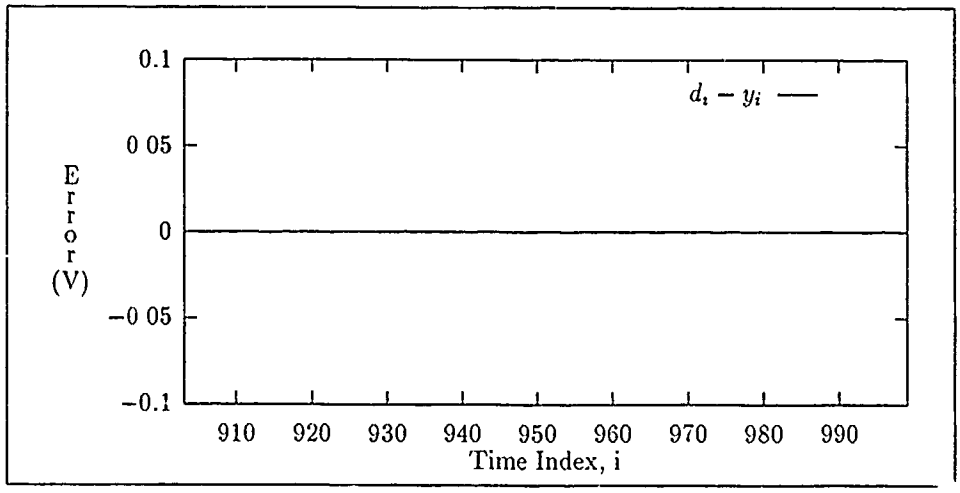

Figure 4.8. WDF2 Filter Verification Test Output Error.

components. After the first block ($k = 0$), the bin positions 1,2,5,6,9,8,13, and 14 become zero. The bin taps in those positions update after the first block is processed and then freeze because their corresponding inputs are zero value thereafter. Figures 4.17 thru 4.32 illustrate the adaptation tracks for each of the bins utilizing a constant μ value. Table 4.4 contains the experimental versus theoretical tap value results derived using a constant μ value for the bin taps. Initializing the previous input block to the sequence specified in Equation 4.6 produces the spectrum in Figure 4.5. With the initial previous input vector initialized in this fashion, the experimental tap values correspond exactly to the theoretical values.

Alternatively, using an independantly derived μ value for each bin tap, and ignoring the first block the filter processes when accessing the average power in each bin, produces experimental tap values that are equal to the theoretical values. The first previous block, when initialized to zero, generally produces a block spectrum that is unique with respect to succeeding block spectrums. Since the initial block is generally unique it can be considered an anomaly and therefore ignored. The tables and figures displayed for this test are otherwise representative of the results obtained when using a different μ value for each bin tap. Clearly, the figures and tap adaptation results show that WLMS2 is a valid Walsh-domain weight-update equation.

Tap	*Theoretical*	*Experimental*
0	1.234	1.234
1	0	0.123
2	0	0.061
3	0.612	0.612
4	3.512	3.512
5	0	0.351
6	0	0.029
7	0.292	0.292
8	4.823	4.823
9	0	0.482
10	0	0.191
11	1.914	1.914
12	2.831	2.831
13	0	0.283
14	0	0.293
15	2.931	2.931

Table 4.4. WDF2 Verification Test Experimental Results

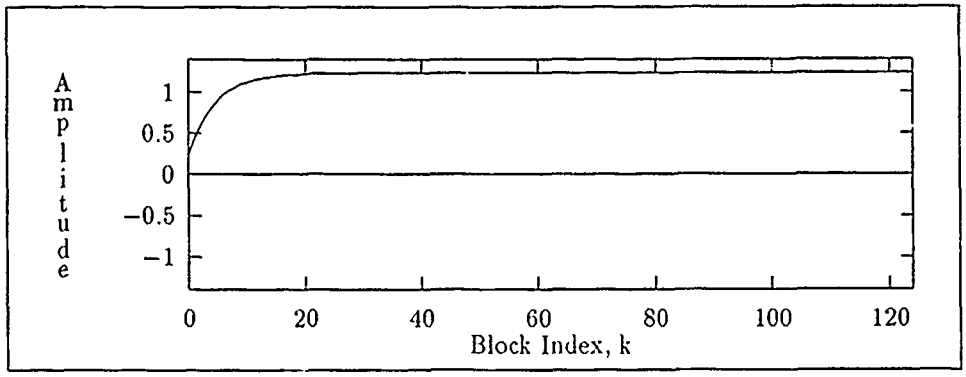

Figure 4.9. WDF1 $H_0(k)$ Adaptation Track

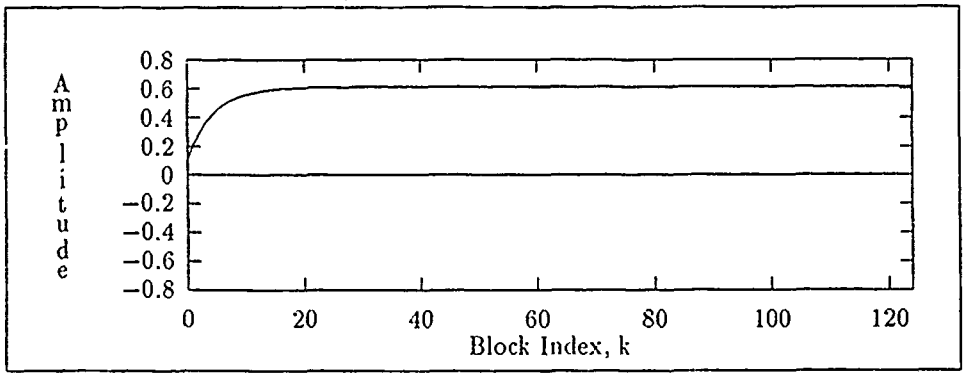

Figure 4.10. WDF1 $H_1(k)$ Adaptation Track

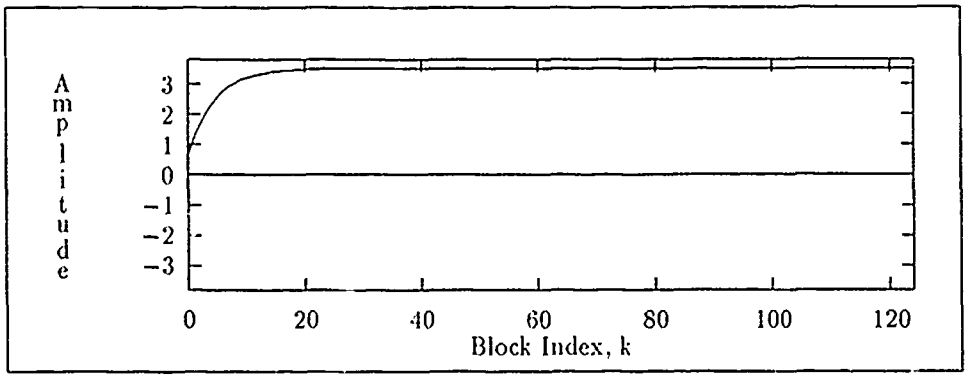

Figure 4.11. WDF1 $H_2(k)$ Adaptation Track

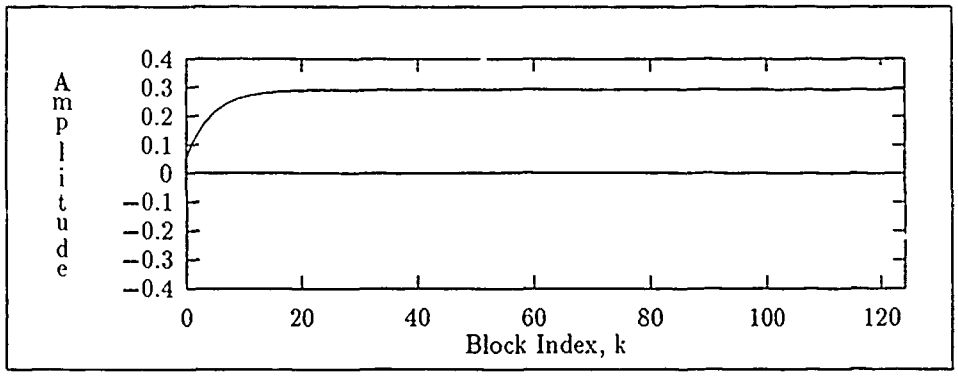

Figure 4.12. WDF1 $H_3(k)$ Adaptation Track

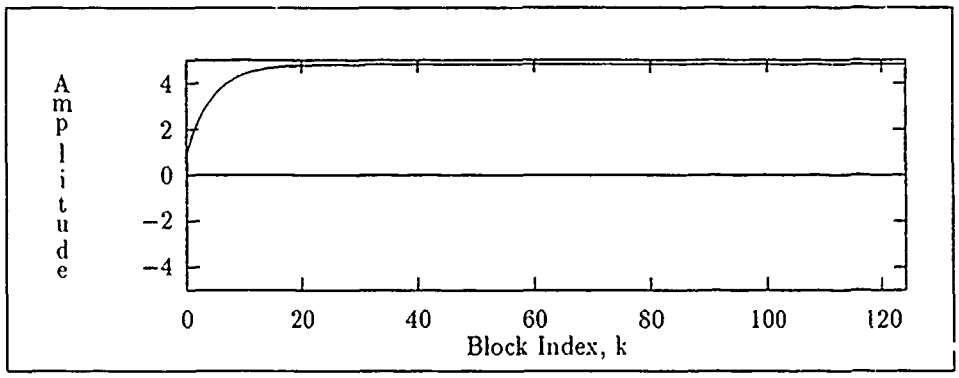

Figure 4.13. WDF1 $H_4(k)$ Adaptation Track

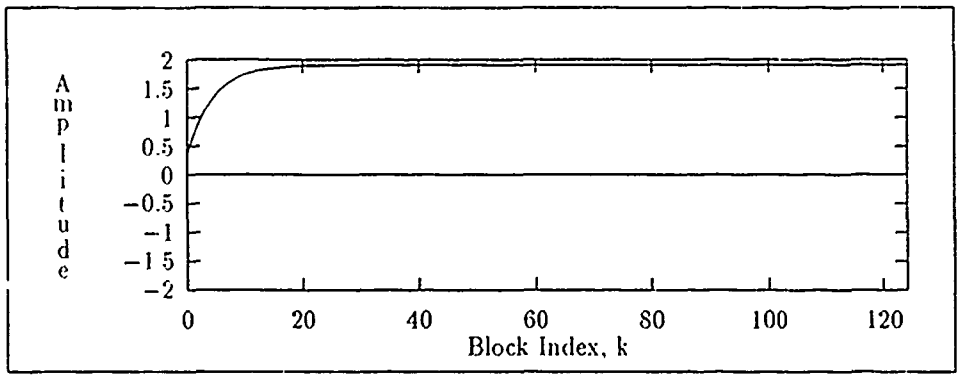

Figure 4.14. WDF1 $H_5(k)$ Adaptation Track

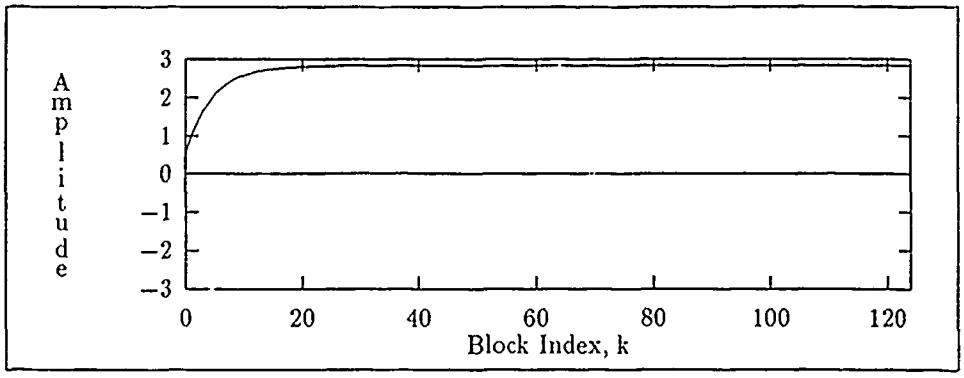

Figure 4.15. WDF1 $H_6(k)$ Adaptation Track

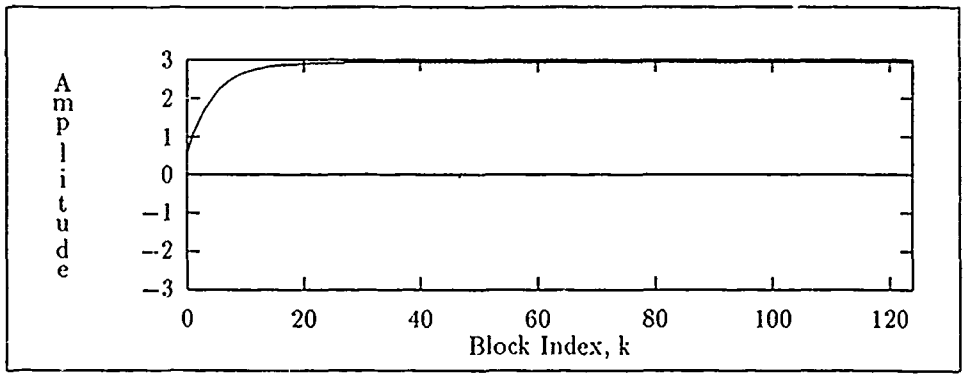

Figure 4.16. WDF1 $H_7(k)$ Adaptation Track

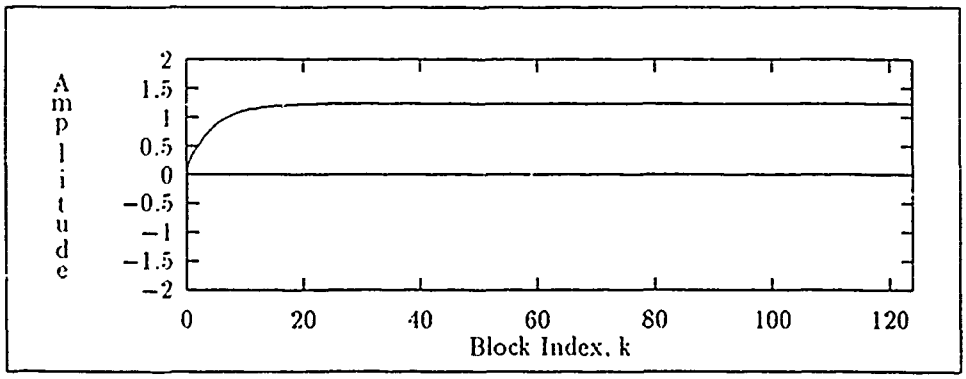

Figure 4.17. WDF2 $H_0(k)$ Adaptation Track

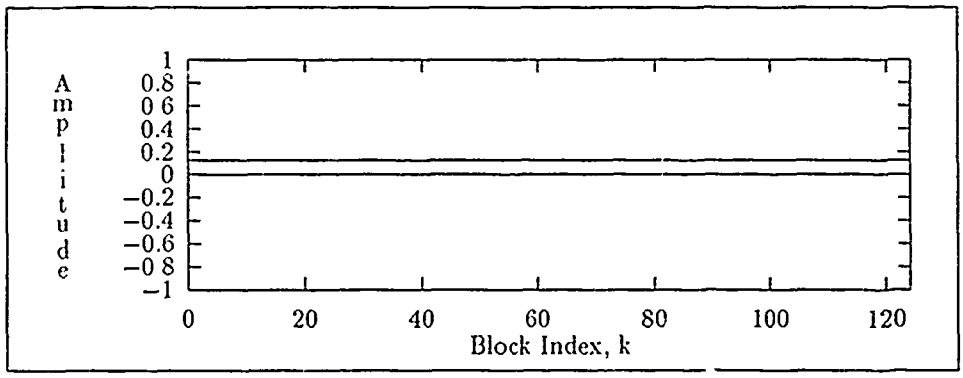

Figure 4.18. WDF2 $H_1(k)$ Adaptation Track

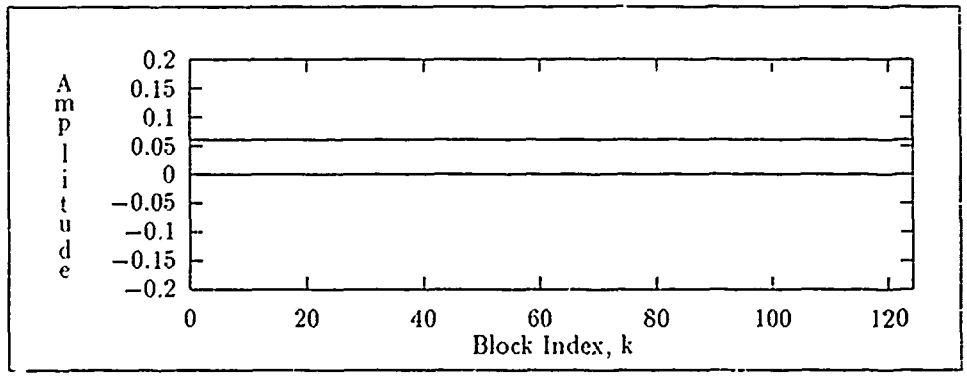

Figure 4.19. WDF2 $H_2(k)$ Adaptation Track

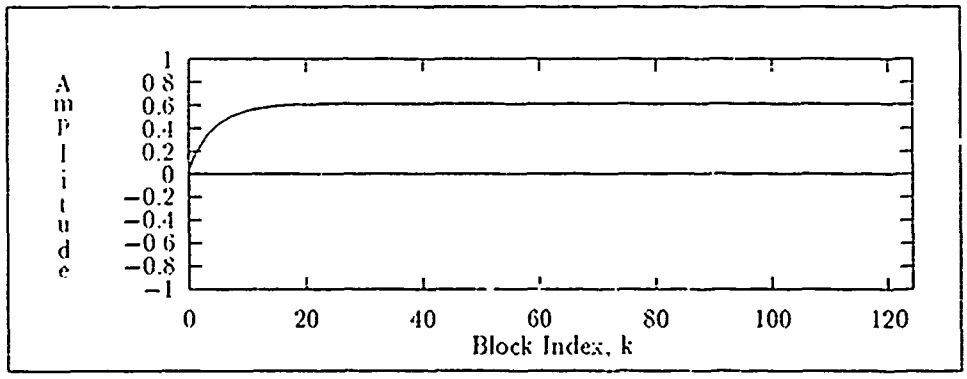

Figure 4.20. WDF2 $H_3(k)$ Adaptation Track

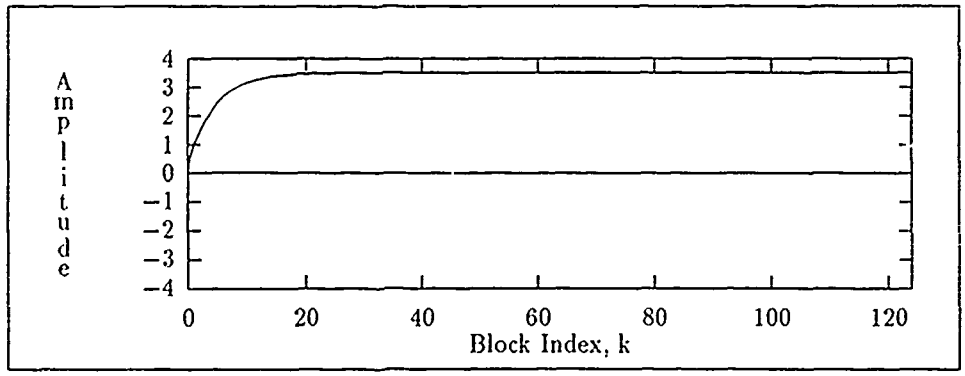

Figure 4.21. WDF2 $H_4(k)$ Adaptation Track

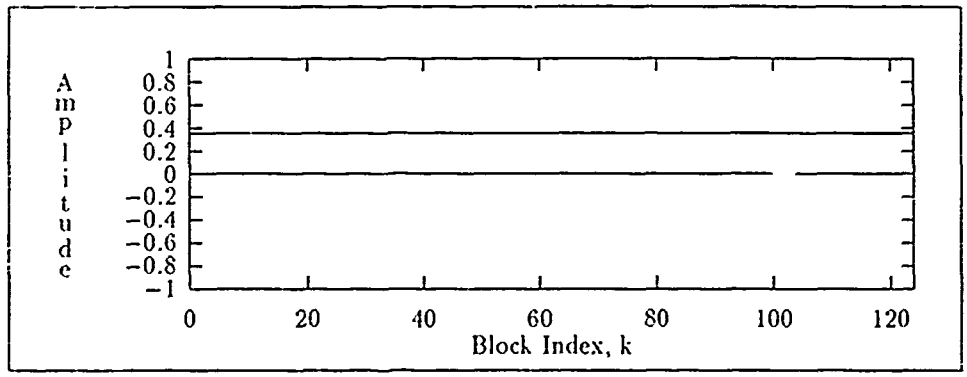

Figure 4.22. WDF2 $H_5(k)$ Adaptation Track

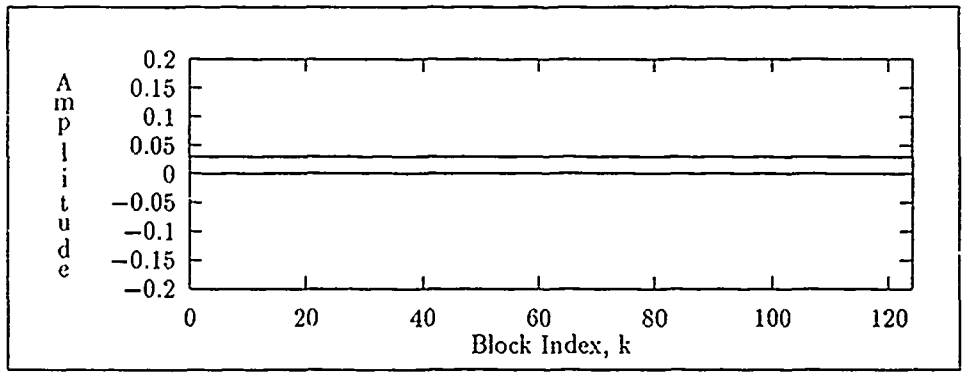

Figure 4.23. WDF2 $H_6(k)$ Adaptation Track

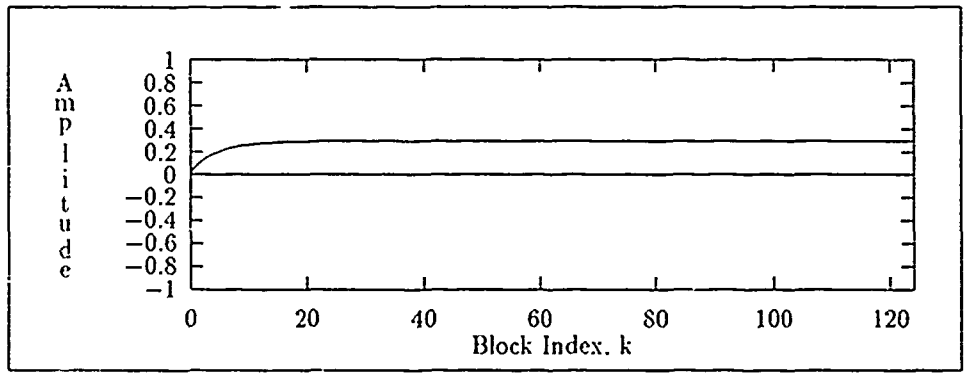

Figure 4.24. WDF2 $H_7(k)$ Adaptation Track

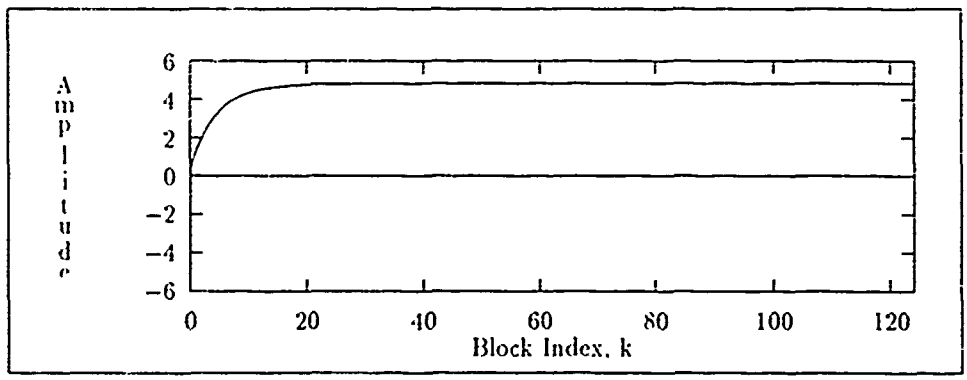

Figure 4.25. WDF2 $H_8(k)$ Adaptation Track

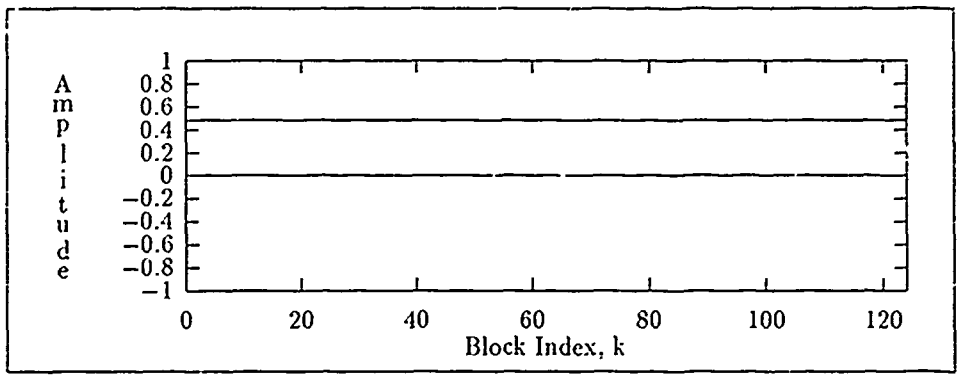

Figure 4.26. WDF2 $H_9(k)$ Adaptation Track

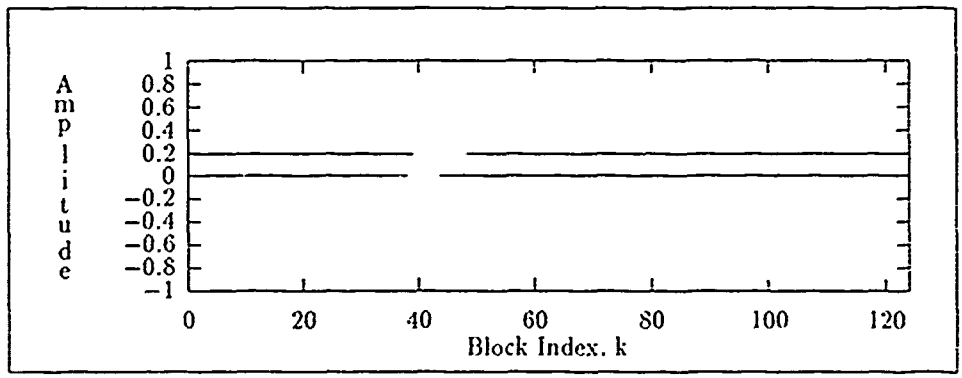

Figure 4.27. WDF2 $H_{10}(k)$ Adaptation Track

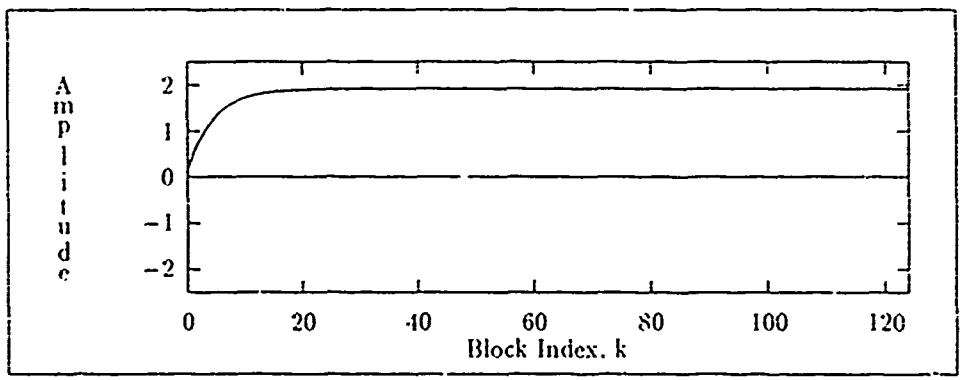

Figure 4.28. WDF2 $H_{11}(k)$ Adaptation Track

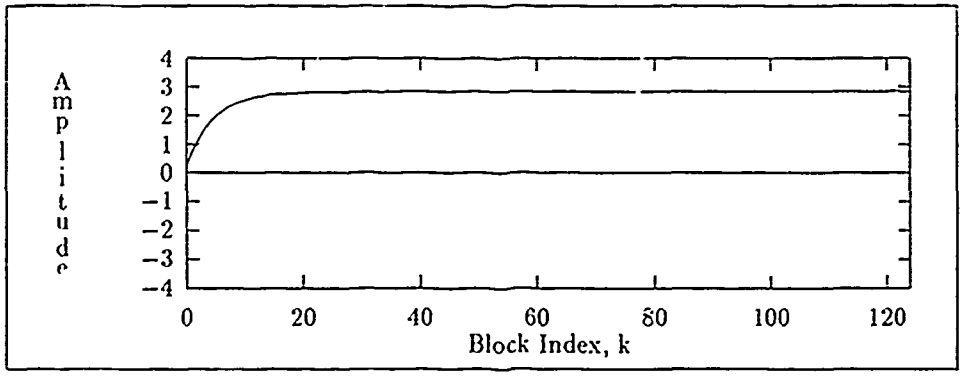

Figure 4.29. WDF2 $H_{12}(k)$ Adaptation Track

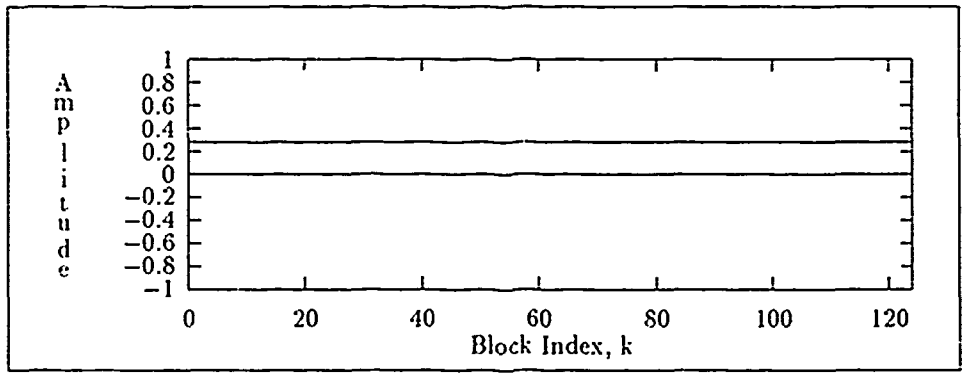

Figure 4.30. WDF2 $H_{13}(k)$ Adaptation Track

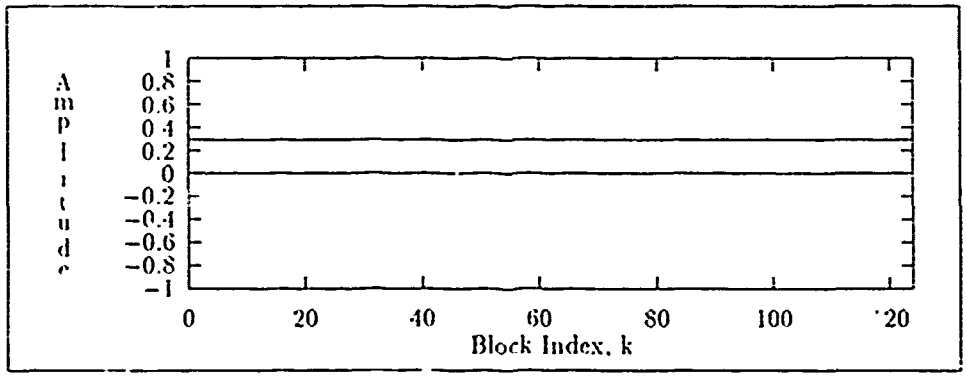

Figure 4.31. WDF2 $H_{14}(k)$ Adaptation Track

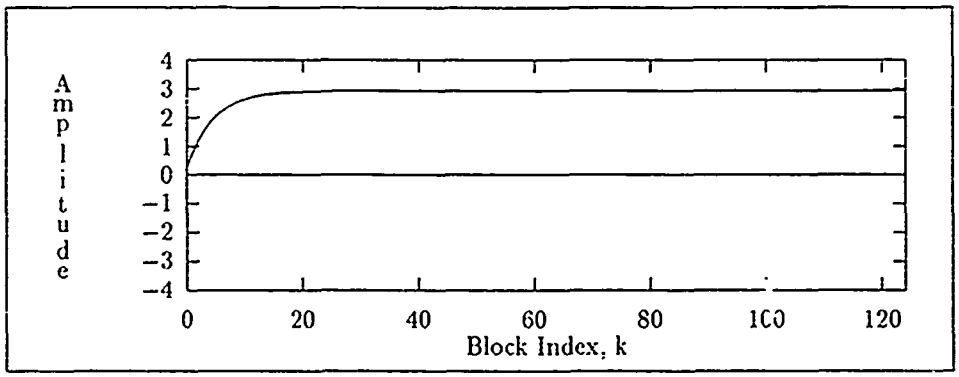

Figure 4.32. WDF2 $H_{15}(k)$ Adaptation Track

4.3 Filter Verification Test 2

This section addresses a test that was used to validate the TDF and FDF software. This test was also used on the WDF1 and WDF2 filters and the results are discussed.

The FDF and TDF filters were tested using the filters in a forward modeling configuration (See Figure 4.33). The input is zero-mean unit-variance Additive White Gaussian Noise (N(0,1)). The desired signal is the N(0,1) noise passed through the plant (a four tap digital FIR filter). Utilizing the Z-transform, the theoretical transfer function of the adaptive filters after convergence is given by [10]

$$H(z) = \frac{\Phi_{xd}(z)}{\Phi_{xx}(z)} \tag{4.14}$$

where $\Phi_{dx}(z)$ is the Cross-power spectrum of the desired signal and input signal and $\Phi_{xx}(z)$ is the Auto-power spectrum of the input signal. Representing the z-transform of the plant transfer function by $H_p(z)$, the the Cross-power spectrum can be expressed as [10]

$$\Phi_{xd}(z) = H_p(z)\Phi_{xx}(z) \tag{4.15}$$

The Auto-power spectrum, $\Phi_{xx}(z)$, in this case is one because the input is N(0,1) noise [10]. Utilizing this fact and Equations 4.14 and 4.15, the transfer function of the adaptive filters is determined by

$$H(z) = \frac{H_p(z)\cdot 1}{1} \tag{4.16}$$

Equation 4.16 states that the transfer functions of the adaptive filters should theoretically be equal to the plant transfer function. In other words, the filter tap weights should converge

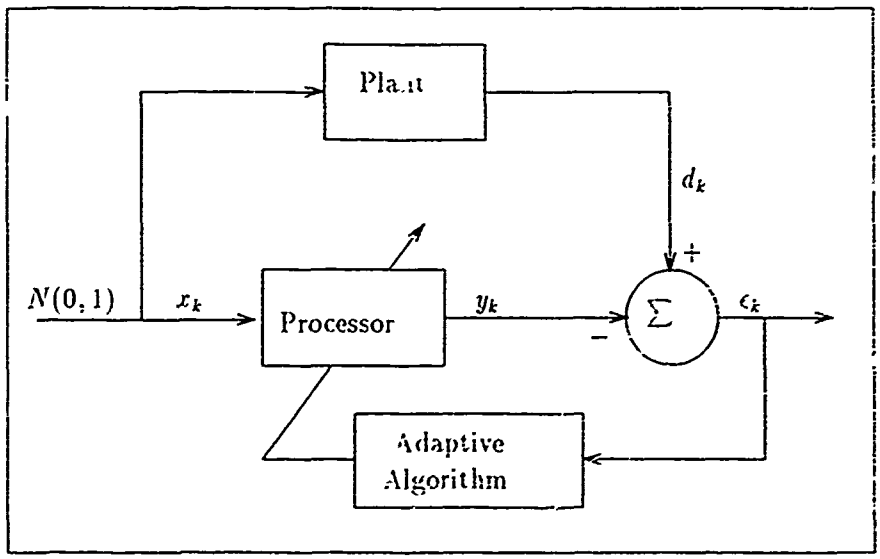

Figure 4.33. Verification Test 2 Configuration

Tap	*Case 1*	*Case 2*
h_0	6.123	6.123
h_1	0	1.127
h_2	0	1.349
h_3	0	4.001

Table 4.5. Forward Modelling Test:Plant tap values for Case 1 and Case 2

to the plant FIR filter tap weights. In the FDF case, this means that the inverse FFT of the frequency-domain bin taps, after they have converged, should be equivalent to the plant FIR filter tap weights. Two sets of tap weights were used for the plant FIR in this test (Table 4.5). The first was a trivial case (See Table 4.5 Case 1): set the h_0 tap to 6.123 and the remaining three to zero. This configuration simply scales the N(0,1) signal by a factor of 6.12. The FDF filter was configured to process 4 point blocks, which means the filter was using 8-point FFTs and, accordingly had 8 frequency-domain taps. The 8 frequency-domain taps inverse transform to an equivalent 4 tap time domain FIR filter via the gradient constraint used in the FDF2 design.

For Case 1, the TDF taps converged to [6.123 0 0 0]. The inverse FFT of the FDF frequency-domain taps, after convergence, produced the same result. In Case 2, the

time-domain tap weights for both filters achieved the plant solution. In both cases, a misadjustment value of $M = 0.1$ was used and the FDF filter used a constant μ value. The FDF filter failed to converge to the theoretical tap values when a separate μ for each frequency bin was used.

The forward modelling test was also used to verify the WDF1 and WDF2 filters. Both filters were configured to process 4-point blocks, using 4 and 8 Walsh-domain taps for the WDF1 and WDF2 filters respectively. The equivalent time-domain dyadic convolution filter for WDF1 has 4 taps and 8 taps for the WDF2 filter, because WDF2 doesn't use a gradient constraint. Neither filter was expected to achieve the plant values because both filters perform dyadic convolution. In this case, the N(0,1) input and the plant filtered version of the N(0,1) signal serve as a nonperiodic input and desired signal pair.

Using the tap values for Case 1, and a Misadjustment of $M = 0.1$, all of the Walsh-domain tap values converged to 6.123 for the WDF1 and WDF2 filters. Both the independant and constant bin μ calculations produced the same result. This matches the theoretical expectation. N(0,1) noise produces an N-point DWT composed of all N sequency functions used in the transform. The DWT is a linear transform (Section 2.1.2), so the scaling performed by the Plant results in a scaled version of the input signal transform for the desired signal. Accordingly, the Walsh-domain bin weights should all converge to the scaling factor of 6.123.

For Case 2 and $M = 0.1$; neither WDF1 or WDF2 were able to filter the N(0,1) noise to produce the Plant output. This was true for both an independant and constant bin μ. For both filters, the Walsh tap adaptation tracks were excessively noisy and, with the exception of the zero sequency bin tap, failed to converge. Figures 4.34 thru 4.39 show the adaptation tracks for the WDF2 filter bin taps $H_0(k)$, $H_5(k)$, and $H_2(k)$; as well as the respective input signal transform component to desired signal transform component ratios. These three bin taps represent best, typical, and worst case in terms of adaptation noise present.

As the figures show, tap $H_0(k)$ has the least amount of adaptation noise. Comparing the bin ratios for the three taps (Figures 4.35, 4.37, and 4.39) clearly shows that the adaptation noise for each tap is proportional to the variation in the bin ratio, where the nth bin ratio is defined

$$Ratio_n(k) = \frac{X_n(k)}{D_n(k)} \tag{4.17}$$

where k represents the block index and $k = 0, 1, \ldots$. The perturbation in Figure 4.37 at approximately $k = 110$ is due to a very small $D_5(110)$ value. The ratio variation is due

to the fact that the filtered noise is also Gaussian but uncorrelated with the N(0,1) input, so the resulting spectrums from block to block will differ between input and desired. The tap adaptation tracks and bin ratios shown for the WDF2 filter are representative of results produced by the WDF1 filter.

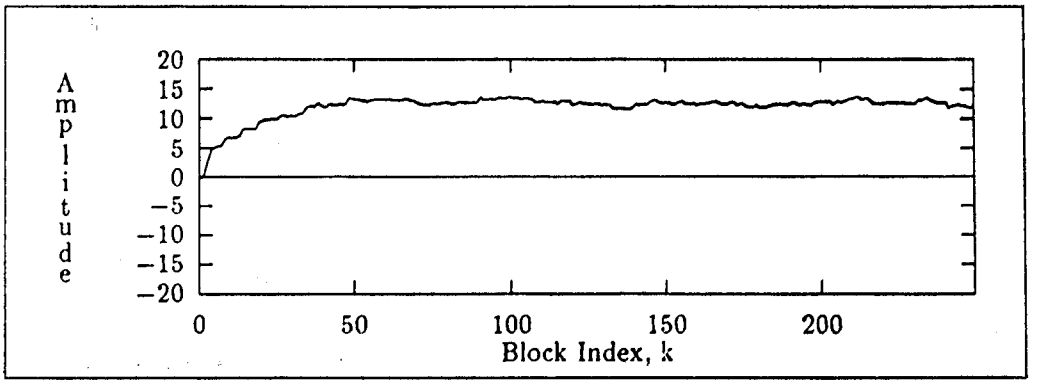

Figure 4.34. WDF2 filter tap $H_0(k)$: $H_0(k)$ adaptation track for noise input and filtered noise desired signal using constant bin μ and $M = 0.1$.

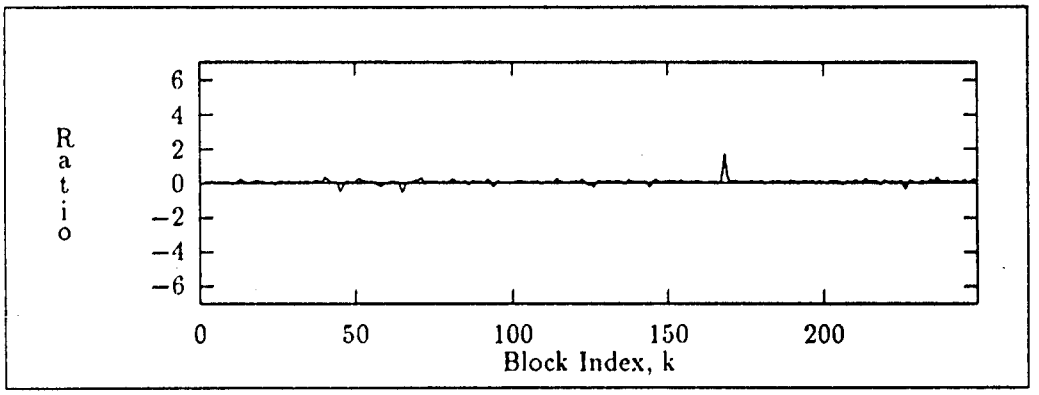

Figure 4.35. WDF2 $Ratio_0(k)$. This is the ratio of the noise input 8-point DWT bin 0 and the filtered noise desired signal 8-point DWT bin 0 versus k.

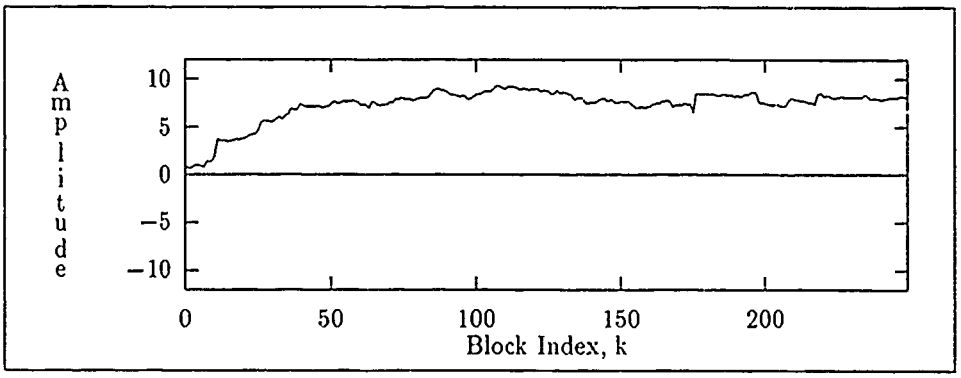

Figure 4.36. WDF2 filter tap $H_5(k)$: $H_5(k)$ adaptation track for noise input and filtered noise desired signal using constant bin μ and $M = 0.1$.

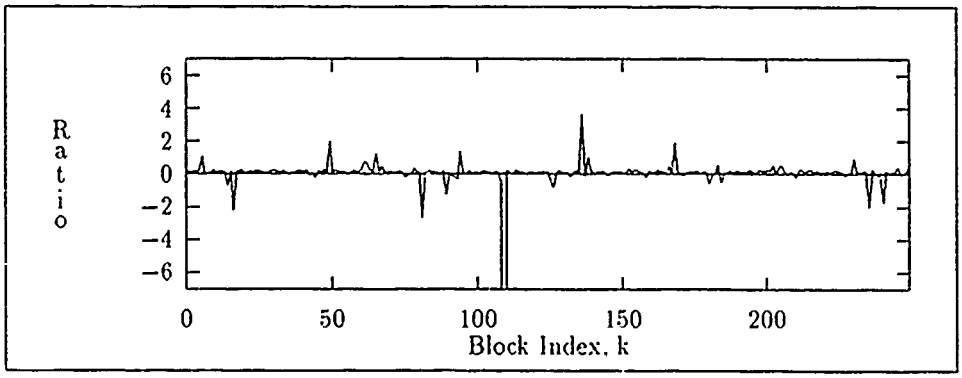

Figure 4.37. WDF2 $Ratio_5(k)$. This is the ratio of the noise input 8-point DWT bin 5 and the filtered noise desired signal 8-point DWT bin 5 versus k.

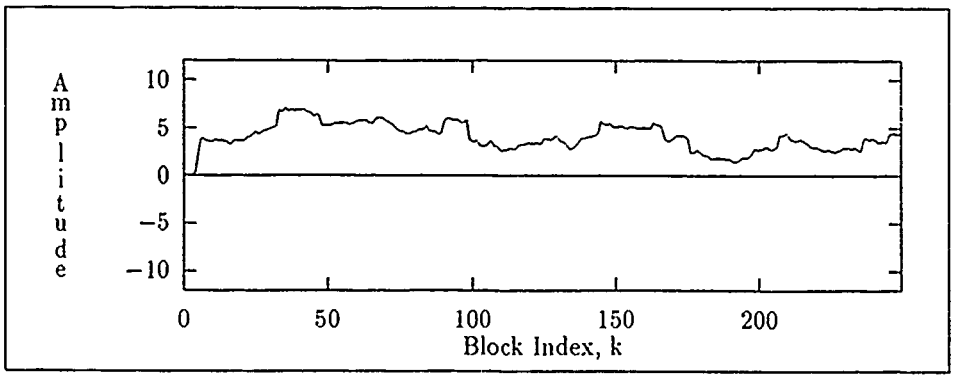

Figure 4.38. WDF2 filter tap $H_2(k)$: $H_2(k)$ adaptation track for noise input and filtered noise desired signal using constant bin μ and $M = 0.1$.

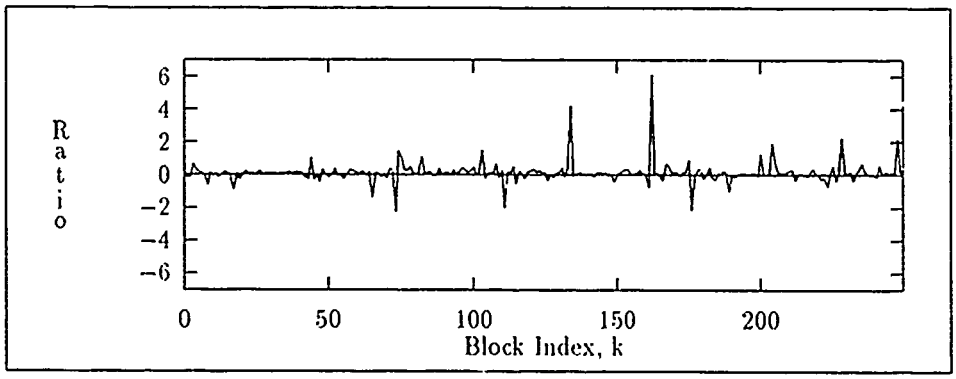

Figure 4.39. WDF2 $Ratio_2(k)$. This is the ratio of the noise input 8-point DWT bin 2 and the filtered noise desired signal 8-point DWT bin 2 versus k.

4.4 Single Tap Time-domain Filter Test

The result of Verification Test 2 (See Section 4.3) prompted the testing of a single tap time-domain filter. The hypothesis is that the dynamics of a single tap time-domain LMS filter (See Equation 2.34) are an effective model of the dynamics of the WDF1 and WDF2 real bin taps. This hypothesis is important because it provides some measure of predictability for the WDF1 and WDF2 filtering performance. Figure 4.40 shows the test configuration. $N(0,1)$ noise is filtered by a single tap Infinite Impulse Response (IIR) filter and a constant is added to the result.

The purpose of the noise filtering procedure is to create a signal with a slowly varying random envelope. The resultant signal serves as the input signal, x_i (See Figure 4.41), to a single tap time-domain adaptive LMS filter. The desired signal is a constant and the gain constant was $\mu = 0.01$. Figure 4.42 shows the $h_0(i)$ tap adaptation track versus the discrete time index i and Figure 4.43 shows the desired signal to input signal ratio, $Ratio_{dx}(i)$, versus i. Clearly, the adaptation track of the filter tap is characteristic of the noiselike variation of $Ratio_{dx}(i)$. This result is similar to the plots presented in Section 4.3 and therefore supports the hypothesis of this test.

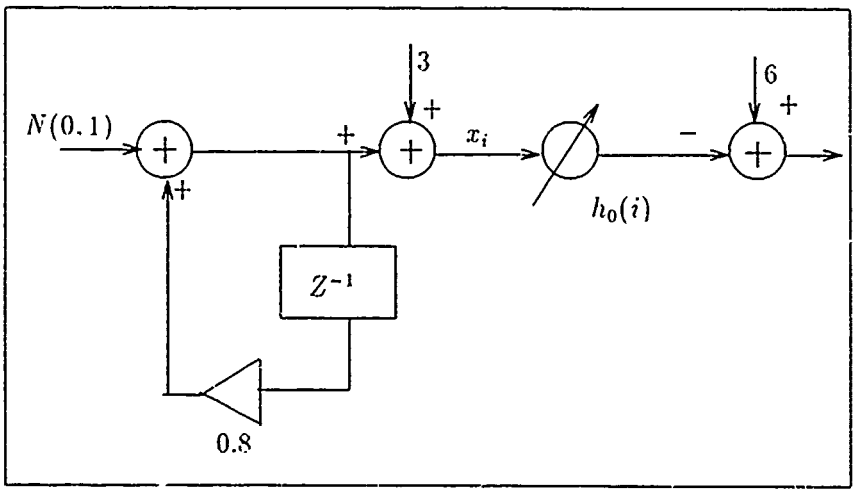

Figure 4.40. Single Tap Time-domain Filter Configuration.

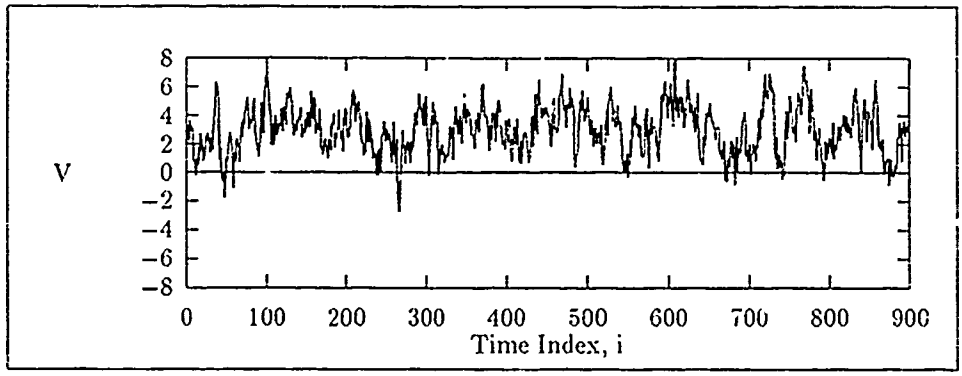

Figure 4.41. Single Tap Time-domain Filter Test: x_i. This is the filtered noise input.

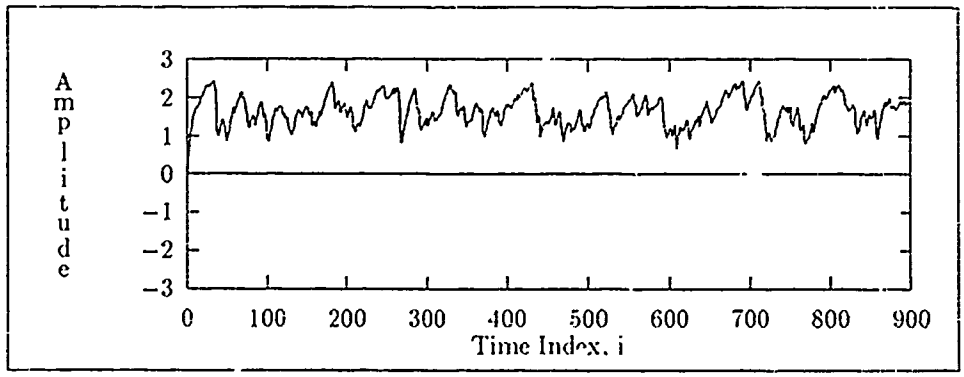

Figure 4.42. Single Tap Time-domain Filter Test: $h_0(i)$. This is the $h_0(i)$ adaptation track versus i.

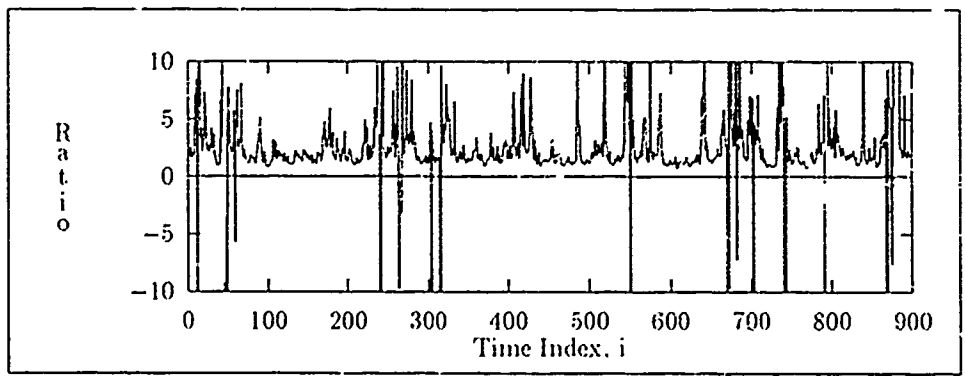

Figure 4.43. Single Tap Time-domain Filter Test: $Ratio_{dx}(i)$. This is the ratio of the desired constant to the filtered noise input signal versus i.

4.5 Chapter Summary

The main purpose of this chapter was to verify the WDF1 and WDF2 software implementations. Two tests were used.

The first test used a periodic input and desired signal (See Sections 4.2.3.2 and 4.2.4.2). The input DWT for each block consisted of components that were all equal to 1. The desired DWT for each block consisted of distinct magnitude nonzero components. In filtering the input, the Walsh-domain taps for both filters converged to the corresponding desired transform component values, i.e. the theoretical solution.

The second test validated the TDF and FDF filters, which are used for comparison purposes in this thesis (See Section 4.3). This test used N(0,1) noise as the input signal and a Plant-filtered version of the signal as the desired. Theoretically, the TDF and FDF filter time-domain impulse responses should converge to the Plant tap values, given that the Plant was a 4 tap FIR filter. In both cases the experimental results matched the theoretical prediction.

The last test was also used on the WDF1 and WDF2 filters as a nonperiodic signal test (See Section 4.3). Both the WDF1 and WDF2 filters perform dyadic convolution and therefore were not expected to achieve the impulse response of the Plant, which neither did. Two FIR filter cases were used. The first was a trival case, with only the h_0 tap nonzero the Walsh-domain taps were expected to converge to h_0. Since the DWT of N(0,1) noise produces a DWT with a nonzero spectral component at every position, and the desired spectrum for this case is an h_0 scaled version of the input. Experimentally, both filters achieved the theoretical solution.

For the second case, all four FIR taps were nonzero (See Section 4.3). Experimentally the WDF1 and WDF2 filters were unable to filter the input to produce the desired signal. An analysis of the bin tap adaptation tracks showed excessive adaptation noise that corresponded with the amount of fluctuation that exists in the ratio of the corresponding input transform bin to the desired transform bin from block to block. This prompted a single tap time-domain filter test (See Section 4.4). This test was not comprehensive but clearly indicated that the adaptation dynamics of the WDF1 and WDF2 Walsh-domain taps can be generally modeled by the adaptation dynamics of a single tap time-domain filter.

In summary, the tests in this chapter verified the software implementations of the WDF1 and WDF2 filters. The second test did expose a possible filtering limitation, the Walsh-domain tap adaptation tracks are disrupted by transform component ratio fluctuations

between corresponding input and desired DWT components. The next chapter presents the time-shifted sinusoidal and rectangular signal tests used to establish the filtering performance of the Walsh-domain filters relative to each other and the TDF and FDF filters.

V. *Filter Testing and Comparison*

5.1 Introduction

The previous chapter discussed the software implementation and verification of two Walsh-domain adaptive filters: WDF1 and WDF2. Now that software verification has been accomplished, a relative measure of performance must be established between WDF1 and WDF2. Also, a relative measure of performance must be established between the two Walsh-domain filters and other adaptive filters. Therefore, there are two major goals:

1. The WDF1, WDF2, TDF, and FDF filters are compared in terms of convergence speed and output error using time-shifted noiseless and noisy periodic signals (Section 5.2).
2. A processing speed performance comparison is made between the WDF1,WDF2, TDF, and FDF filters. (Section 5.3).

5.2 Time-shifted Signal Tests

The purpose of this section is to investigate the effects of the DWT's lack of time-shift invariance (Section 2.1.3) on the filtering abilities of the WDF1 and WDF2 filters. Four signal tests are used to conduct the investigation:

1. Signal Test 1 (Section 5.2.1) uses a noiseless periodic rectangular signal as the input signal.
2. Signal Test 2 (Section 5.2.2) uses a noisy periodic rectangular signal as the input signal.
3. Signal Test 3 (Section 5.2.3) uses a noiseless periodic sinusoidal signal as the input signal.
4. Signal Test 4 (Section 5.2.4) uses a noisy periodic sinusoidal signal as the input signal.

For each of the two periodic signals, input and desired are derived from the same signal and the signal period is 16 data points. For the noiseless case, the input is shifted relative to the desired signal. For the noisy case, Additive White Gaussian Noise (N(0,1)) is added to the shifted or unshifted signal to create the input. Progressive input sample shifts are used for each case with a 4 sample shift relative to the desired signal being the maximum. In each test, the signal set is filtered by the WDF1, WDF2, TDF, and the FDF filters. The

TDF filter uses 16 taps while the WDF1, WDF2, and FDF filters use a 16-point block size ($N = 16$). For periodic signals, setting N equal to the period of the signal results in the same input spectrum for each block processed. The filters are then compared in terms of their convergence speed and the amount of error between the filter output and the desired signal for the last 6 blocks processed, which would be 96 output samples.

The number of weight updates required to achieve 10% of the normalized mean-square-error (NMSE) serves as the criteria for the convergence speed comparison. MSE learning curves for signal tests 1 and 3, are derived by squaring the output error. The MSE learning curves for signal tests 2 and 4 are derived using 100 data files with the noise components uncorrelated between files and the ensemble noise components being N(0,1) samples. The MSE learning curves for all 4 signal tests are normalized by the desired signal power.

5.2.1 Signal Test 1 This test evaluates the WDF1 and WDF2 discontinuous signal filtering performance relative to the TDF and FDF filters. A rectangular signal is used as a simple discontinuous signal representative. The hypothesis is that the Walsh-domain filters will converge faster and produce less error filtering rectangular signals than the FDF and TDF filters and that shifting the input will degrade the Walsh-filter performance. There are two attributes of the DWT which suggest this hypothesis:

1. The DWT of a discontinuous signal produces fewer spectral terms than the corresponding DFT spectrum (See Section 2.1.2.2).
2. The DWT spectrum is not time-shift invariant (See Section 2.1.3).

The input signal used in the test is depicted in Figure 5.1. This signal functions as the input and desired signal in this test. A 992 sample datasize is used because it produces an integer number of 16-point blocks, as opposed to a datasize of 1000 which does not. The input signal was time-shifted to assess time-shift affects on filter error performance and convergence speed. An incremental shift of 1 sample is made on each filter run relative to the desired signal, with a maximum relative shift of 4 samples. For the remainder of this test, the input signal for an n-point relative shift is designated the $n - shift$ input. Filter output results for the $1 - shift$, $3 - shift$, and $4 - shift$ input are presented as they represent best, typical, and worst case, in that order.

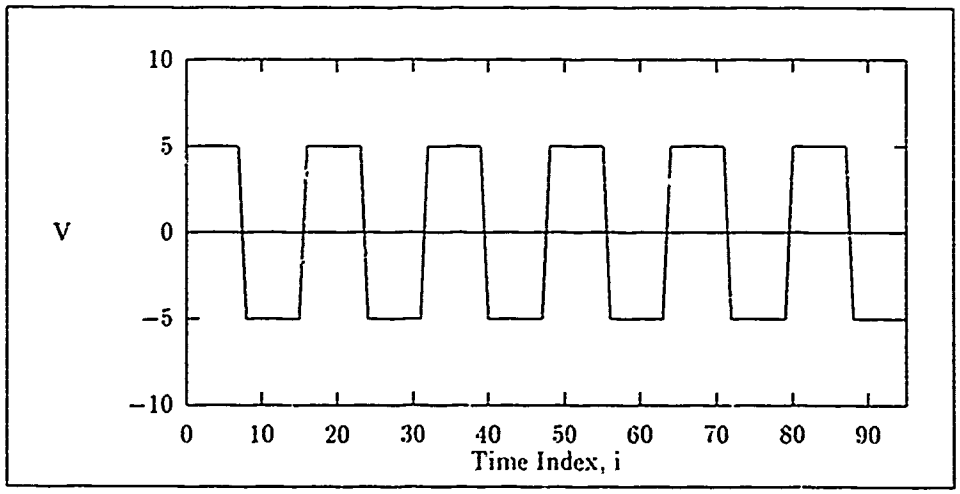

Figure 5.1. Signal Test 1 Rectangular Waveform.

5.2.1.1 Noiseless 1 – shift Results. The 1 – *shift* signal pair is depicted in Figure 5.2. The filter configuration used for all 4 filters is specified in Table 5.1. Figures 5.3, 5.4, 5.5, and 5.6 show the error for the last 96 output samples produced by each filter. Clearly, the Walsh-domain filters produce less error than the FDF and TDF filters for the 1 – *shift* input. The two Walsh-domain filters produce zero error for the last 96 output samples.

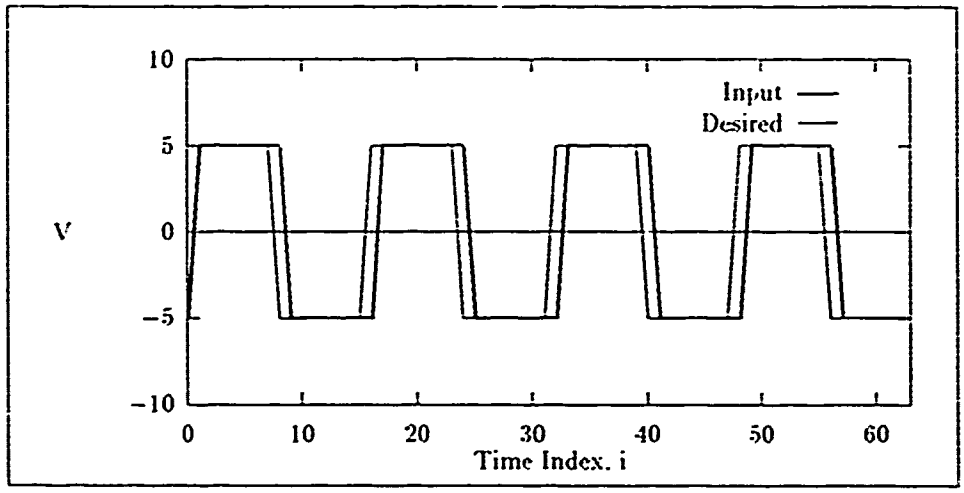

Figure 5.2. Signal Test 1: 1 – *shift* filter inputs. This is the noiseless 1 – *shift* rectangular input and desired signal.

Parameter	*Setting*
Block Size	16
Misadjustment	0.05
Datasize	992

Table 5.1. Signal Test 1: 1 – *shift* input filter settings

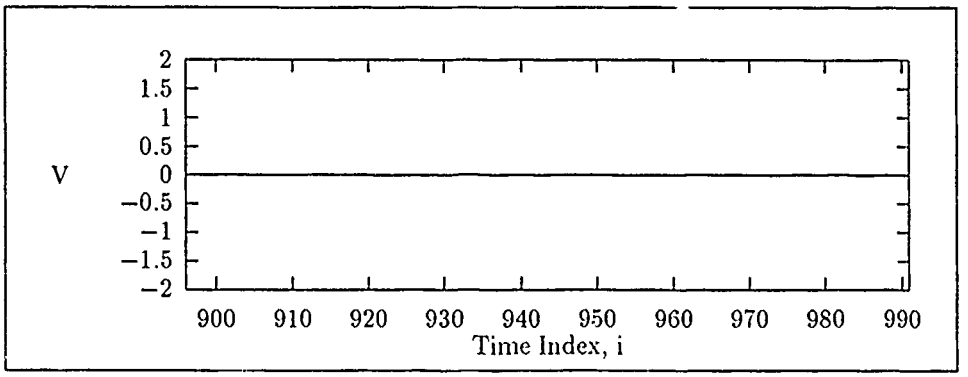

Figure 5.3. Signal Test 1: WDF1 filter output error for $1 - shift$ input. This is the WDF1 output error for the last 96 output samples using $M = 0.5$.

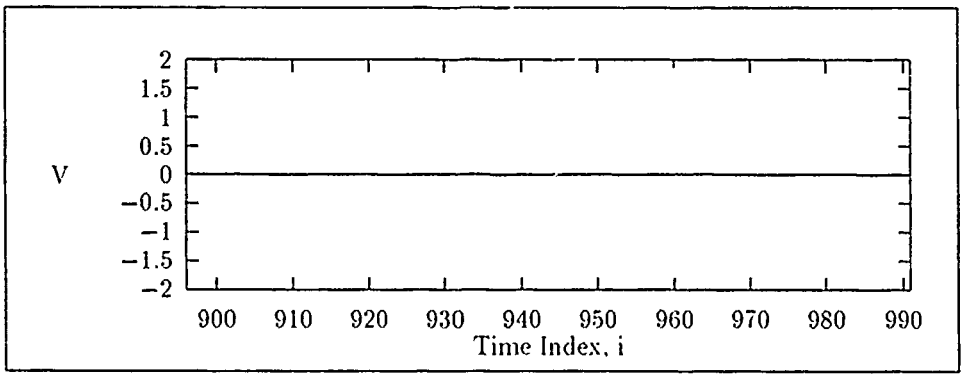

Figure 5.4. Signal Test 1: WDF2 filter output error for $1 - shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.5$.

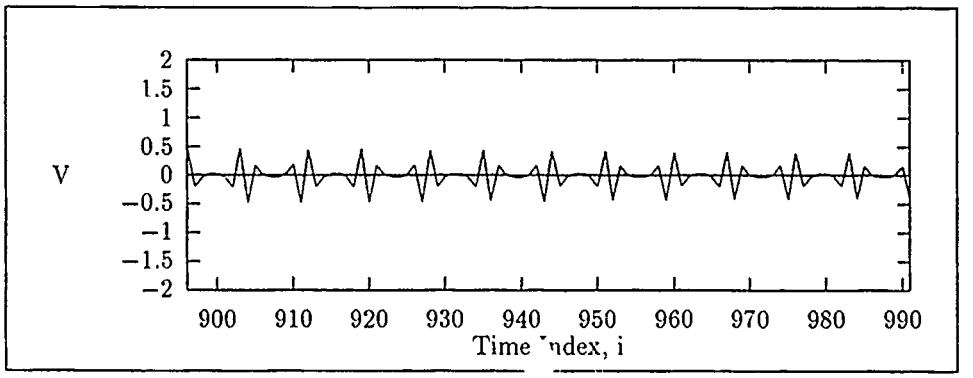

Figure 5.5. Signal Test 1: TDF filter output error for $1 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.5$.

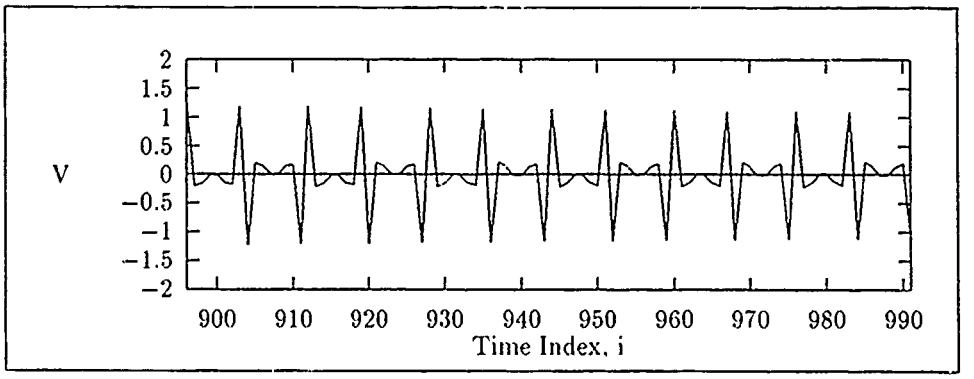

Figure 5.6. Signal Test 1: FDF filter output error for $1 - shift$ input. This is the FDF2 filter output error for the last 96 output samples using $M = 0.5$.

5.2.1.2 Noiseless 3 – shift Results. The $3 - shift$ signal pair is depicted in Figure 5.7. The filter configuration used for all 4 filters is specified in Table 5.2. Figures 5.8, 5.9, 5.10, and 5.11 show the error for the last 96 output samples produced by each filter. Clearly, the Walsh-domain filter output error is less than the TDF and FDF filters for the $3 - shift$ input. The WDF1 and WDF2 $3 - shift$ output error is greater than the $1 - shift$ result.

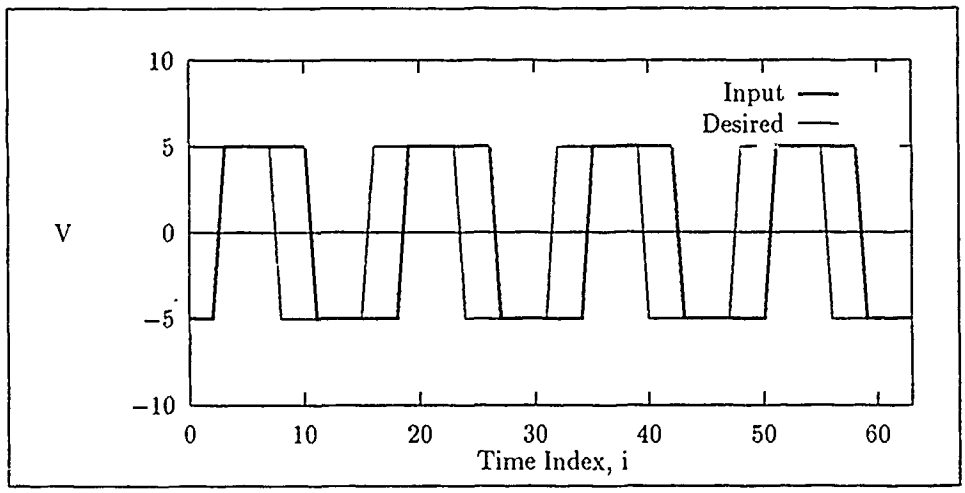

Figure 5.7. Signal Test 1: $3 - shift$ filter inputs. This is the noiseless $3 - shift$ rectangular input and desired signal.

Parameter	*Setting*
Block Size	16
Misadjustment	0.1
Datasize	992

Table 5.2. Signal Test 1: $3 - shift$ input filter settings

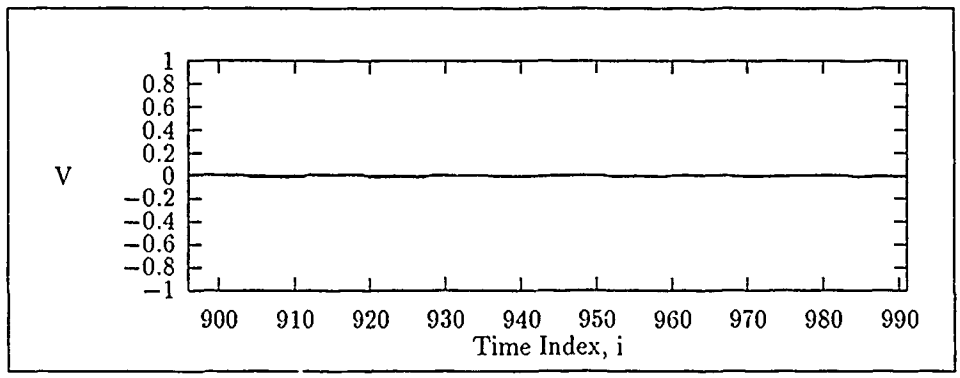

Figure 5.8. Signal Test 1: WDF1 filter output error for $3 - shift$ input. This is the WDF1 output error for the last 96 output samples using $M = 0.1$.

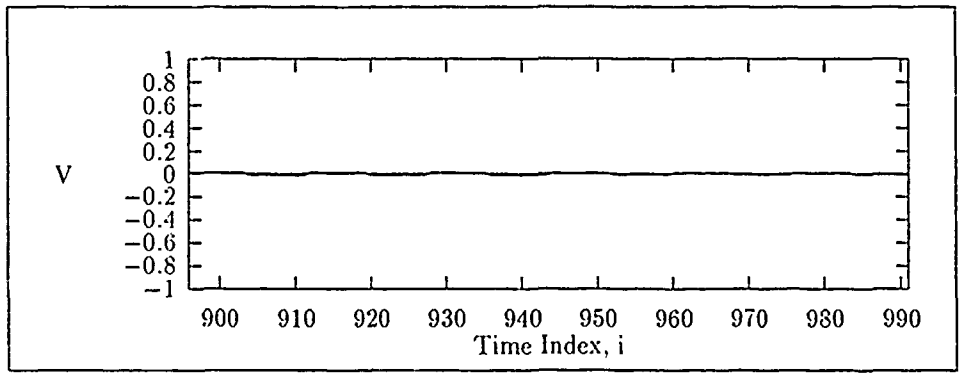

Figure 5.9. Signal Test 1: WDF2 filter output error For $3 - shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.1$.

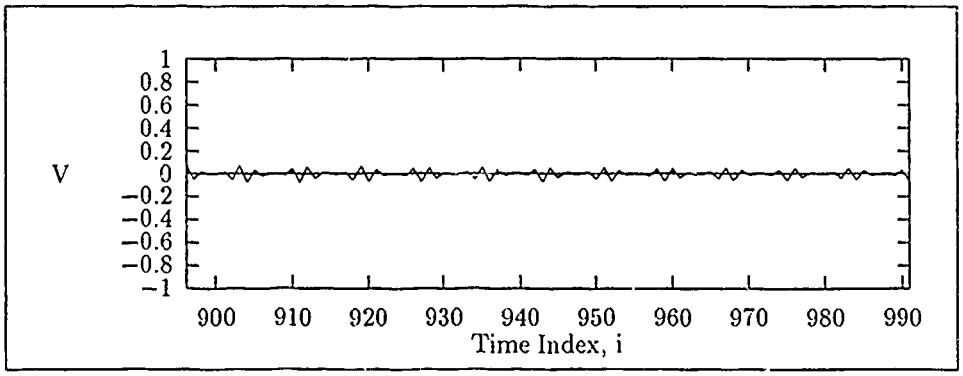

Figure 5.10. Signal Test 1: TDF filter output error for $3 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.1$.

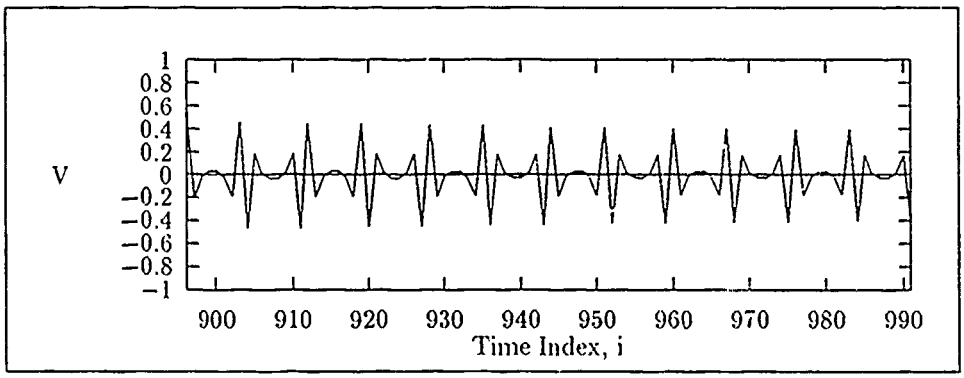

Figure 5.11. Signal Test 1: FDF filter output error for $3 - shift$ input. This is the FDF filter output error for the last 96 output samples using $M = 0.1$.

5.2.1.3 Noiseless $4-shift$ Results. The $4-shift$ signal pair is depicted in Figure 5.12. The filter configuration used for all 4 filters is specified in Table 5.3 while Figures 5.13, 5.14, 5.15, and 5.16 show the error produced for the last 96 output samples. Figures 5.13 and 5.14 show that the Walsh-domain filters were unable to filter the $4-shift$ input.

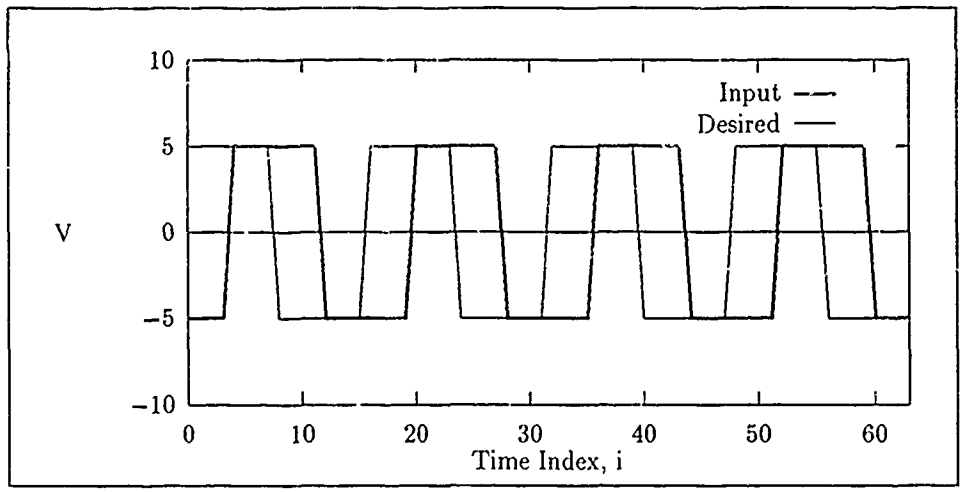

Figure 5.12. Signal Test 1: $4-shift$ filter inputs. This is the noiseless $4-shift$ rectangular input and desired signal.

Parameter	*Setting*
Block Size	16
Misadjustment	0.1
Datasize	992

Table 5.3. Signal Test 1: $4-shift$ filter settings

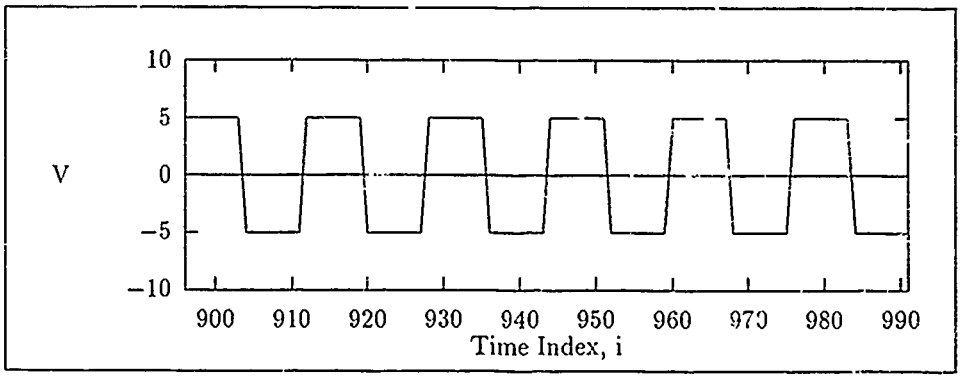

Figure 5.13. Signal Test 1: WDF1 filter output error for $4-shift$ input. This figure depicts the WDF1 output error for the last 96 output samples using $M = 0.1$.

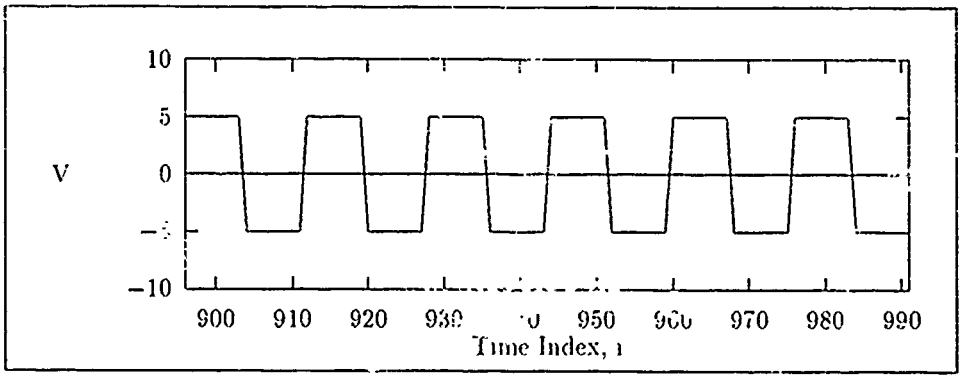

Figure 5.14. Signal Test 1: WDF2 filter output error for $4-shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.1$.

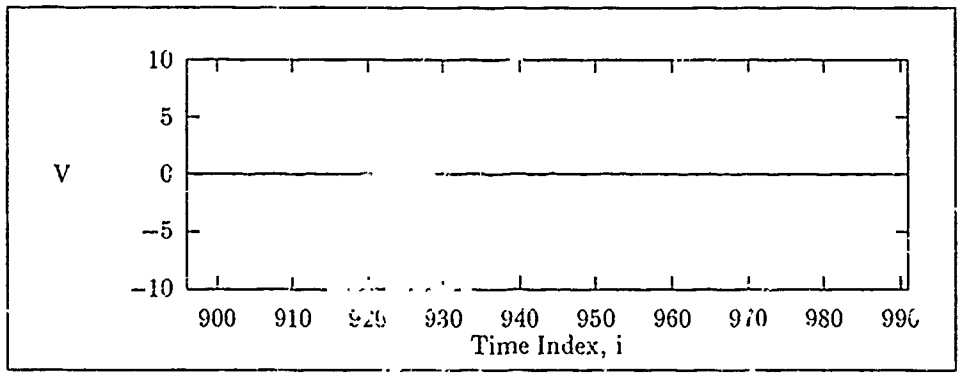

Figure 5.15. Signal Test 1: TDF filter output error for $4 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.1$.

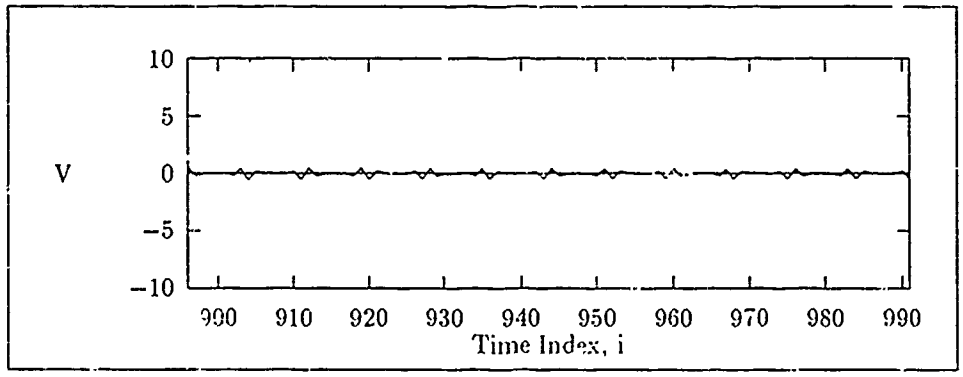

Figure 5.16 Signal Test 1: FDF filter output error for $4 - shift$ input. This is the FDF output error for the last 96 output samples using $M = 0.1$.

5.2.1.4 Signal Test 1 Analysis. For each of the signal sets in Signal Test 1, the three block processing filters performed optimally using a constant μ value, in terms of minimizing the error over the last 96 samples of the 992-point data set. Using an independent μ value for each bin slowed the convergence speed of the filters relative to using a constant μ.

Less error occurs for a constant μ due to the fact that P_{binavg} (See Section 4.2.2) is less than the P_{bin_n} values. Therefore, the independant bin gain constants, μ_n, are less than the constant μ. The Walsh-domain filters' convergence speed decreased as the reference shift increased. The TDF and FDF filters, however, produced output that was essentially

M	*Shift*	*Error Signal Power (Watts)*			
		TDF	FDF	WDF1	WDF2
0.05	1	$5.53E-02$	$3.48E-01$	0	0
0.05	2	$5.04E-02$	$3.33E-01$	0	0
0.1	3	$1.33E-03$	$5.45E-02$	$1.18E-04$	$1.31E-04$
0.1	4	$1.31E-03$	$5.44E-02$	25	25

Table 5.4. Signal Test 1:Error signal power for the last 96 samples.

invariant to the sample shift. The Walsh-domain filters converge more slowly as the shift increases because the input spectrum components become increasingly distinct with respect to the desired signal spectrum (See Section A.3).

Error Performance. For the $1-shift$ and $2-shift$ inputs, the WDF1 and WDF2 filters were able to produce an output that exactly matched the desired signal (Figures 5.3 and 5.4). Conversely, the FDF and TDF filters were still converging at the end of the data set with the TDF filter converging more rapidly (Figures 5.5 and 5.6). Table 5.1 specifies the filter configuration. The FDF filter converged more slowly because the input frequency spectrum is more complex than the sequency spectrum.

Table 5.4 contains the normalized error signal power for each filter, generated from the last 96 error samples produced in each of the four cases. Comparatively, for the first three sample shifts, the Walsh-domain filters' error performances were identical. For the $4-shift$ input, the TDF filter performed slightly better than the FDF filter while the WDF1 and WDF2 filters were unable to filter the 4-sample shifted rectangular signal as Figures 5.11 and 5.13 show. This is due to the fact that the Walsh-domain spectrums for the input and desired are zero magnitude with the exception of one component. The spectral value for the input is not located in the same sequency bin as the desired.

Convergence Speed Comparison. The results of signal test 1 are summarized in Table 5.5 in terms of the number of weight updates required to converge. This table shows in general that the blockprocessing filters require fewer weight updates for this test. With the exception of the $4-shift$ input, the Walsh-domain filters also required the least number of weight updates. The Walsh-domain filter entries for the $4-shift$ input are indicated in the table as "dnc"; indicating "did not converge". Clearly, the WDF1 and WDF2 performances were equivalent.

M	*Shift*	*Number of Weight Updates*			
		TDF	FDF	WDF1	WDF2
0.05	1	297	37	2	3
0.05	2	320	40	6	6
0.1	3	168	21	11	12
0.1	4	168	21	dnc	dnc

Table 5.5. Signal Test 1:Number of weight updates to converge.

Parameter	*Setting*
Block Size	16
Misadjustment	0.05
Datasize	992

Table 5.6. Signal Test 2: $0 - shift$ filter settings.

5.2.2 Signal Test 2 Signal test 2 is an extension of Signal Test 1 (See Section 5.2.1) and evaluates the WDF1 and WDF2 noisy rectangular signal filtering performance relative to the TDF and FDF filters. The WDF1 and WDF2 filter error should increase significantly, relative to the Signal Test 1 results, based on the testing accomplished in Sections 4.3 and 4.4. The input signal used in the test is the signal in Figure 5.1 with noise added. This signal serves as the input and desired signal in this test. The filter input is simply a shifted version of the desired signal. The maximum shift was 4 samples. For the remainder of this test, the input signal for an n-point relative shift is designated the $n - shift$ input. Filter output results are presented for the $0 - shift$, $3 - shift$, and $4 - shift$ input cases, as they represent best, typical, and worst case.

5.2.2.1 Noisy $0 - shift$ *Results.* The filter input signal and desired signal are both depicted in Figure 5.17. The filter configuration used for all 4 filters is specified in Table 5.6 while Figures 5.18, 5.19, 5.20, and 5.21 show the error produced by each filter for the last 96 output samples. The WDF1 and WDF2 error is blocky due to the rectangular nature of the Discrete Walsh functions. Clearly, the Walsh-domain filter error is less than the TDF and FDF error.

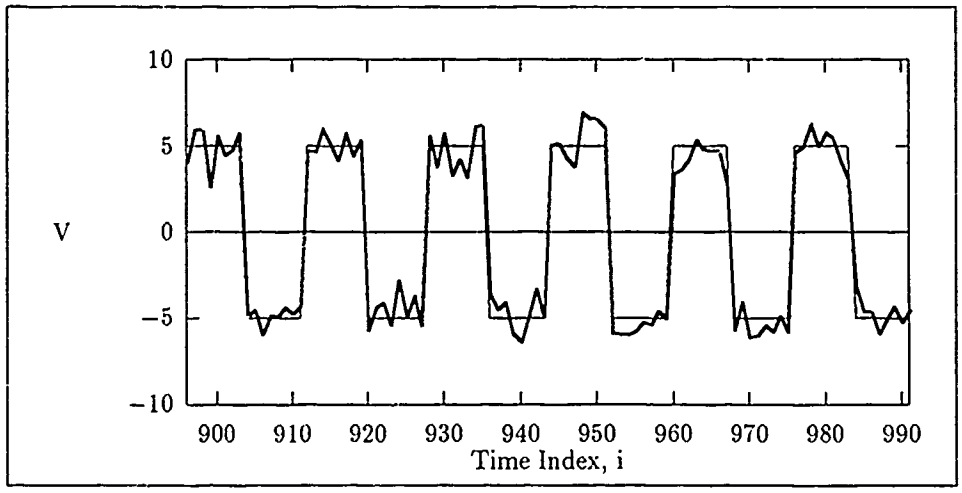

Figure 5.17. Signal Test 2: $0 - shift$ inputs. This is the last 96 samples of the noisy $0 - shift$ rectangular input and desired signal.

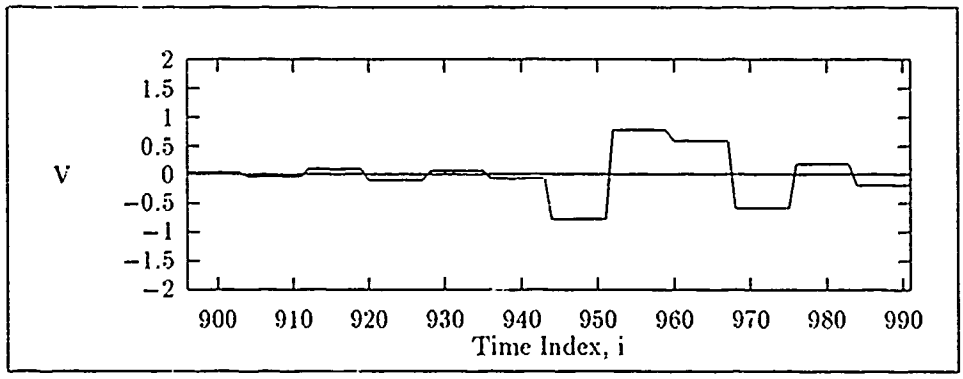

Figure 5.18. Signal Test 2: WDF1 filter output error for $0 - shift$ input. This is the WDF1 output error for the last 96 output samples using $M = 0.05$.

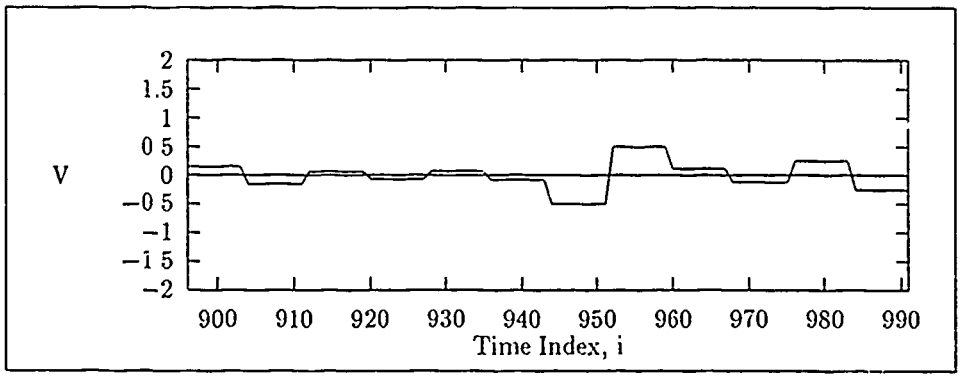

Figure 5.19. Signal Test 2: WDF2 filter output error for $0 - shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.05$.

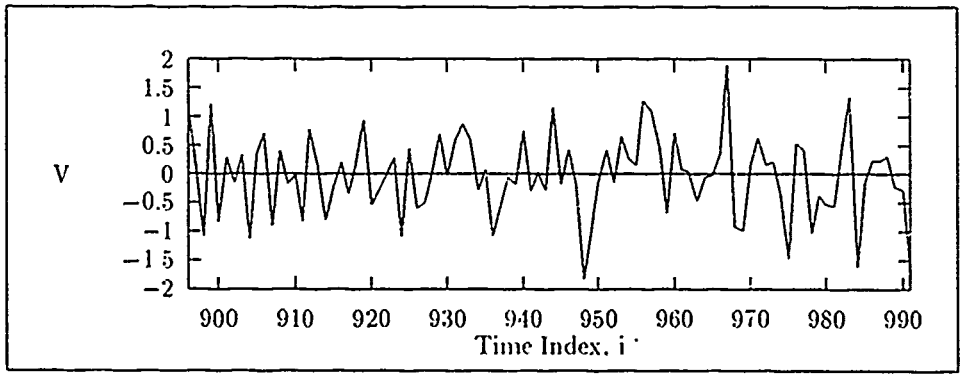

Figure 5.20. Signal Test 2: TDF filter output error for $0 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.05$.

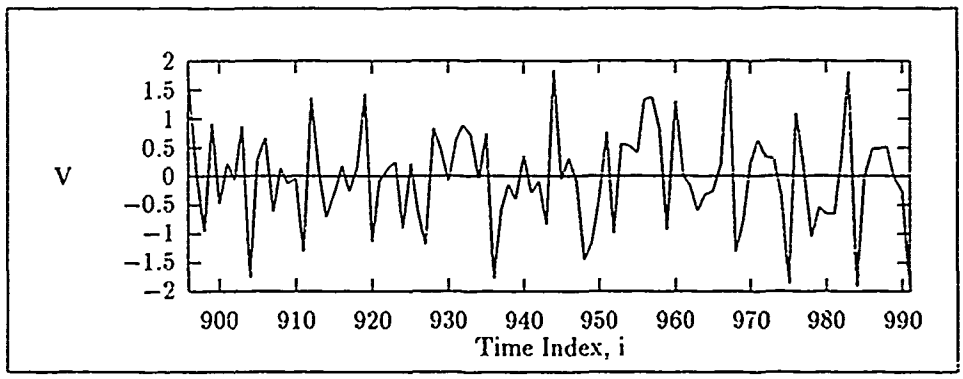

Figure 5.21. Signal Test 2: FDF filter output error for $0 - shift$ input. This is the FDF output error for the last 96 output samples using $M = 0.05$.

5.2.2.2 Noisy 3 – shift Results. The $3 - shift$ signal pair is depicted in Figure 5.22. The filter configuration used for all 4 filters is specified in Table 5.7 while Figures 5.23, 5.24, 5.25, and 5.26 show the error for the last 96 output samples produced by each filter. Clearly, the Walsh-domain filter error has increased relative to the $0 - shift$ results.

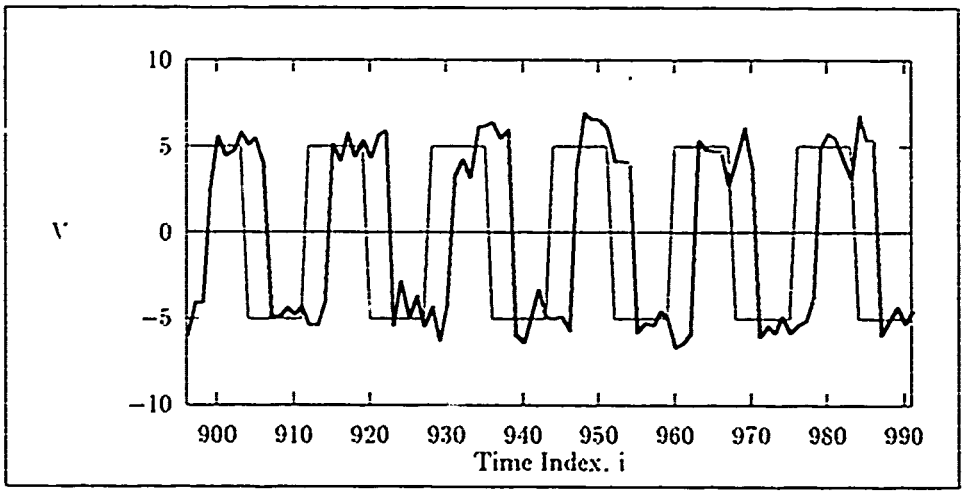

Figure 5.22. Signal Test 2: $3 - shift$ filter inputs. This is the last 96 samples of the noisy $3 - shift$ rectangular input and desired signal.

Parameter	*Setting*
Block Size	16
Misadjustment	0.1
Datasize	992

Table 5.7. Signal Test 2: $3 - shift$ filter settings

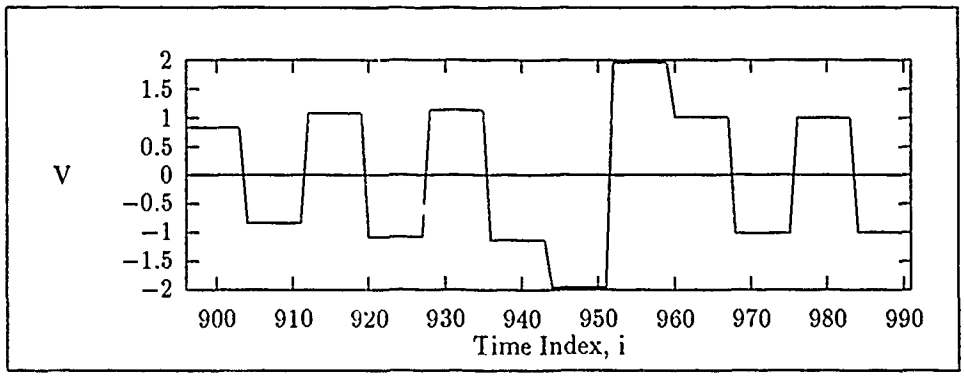

Figure 5.23. Signal Test 2: WDF1 filter output error for $3-shift$ input. This is the WDF1 output error for the last 96 output samples using $M = 0.1$.

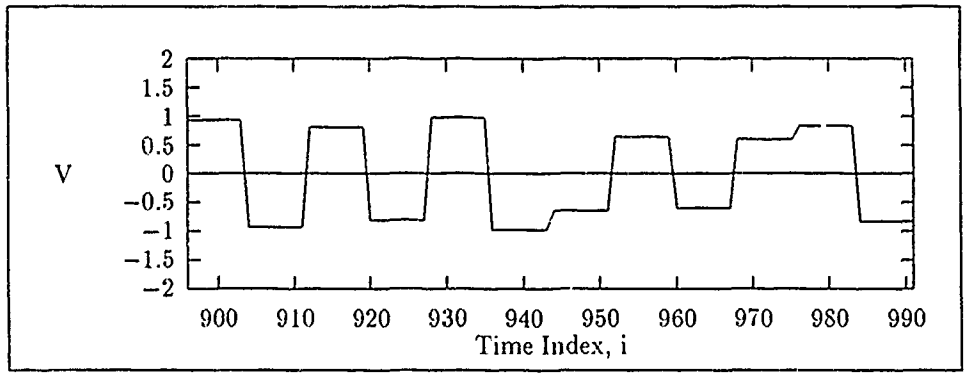

Figure 5.24. Signal Test 2: WDF2 filter output error for $3-shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.1$.

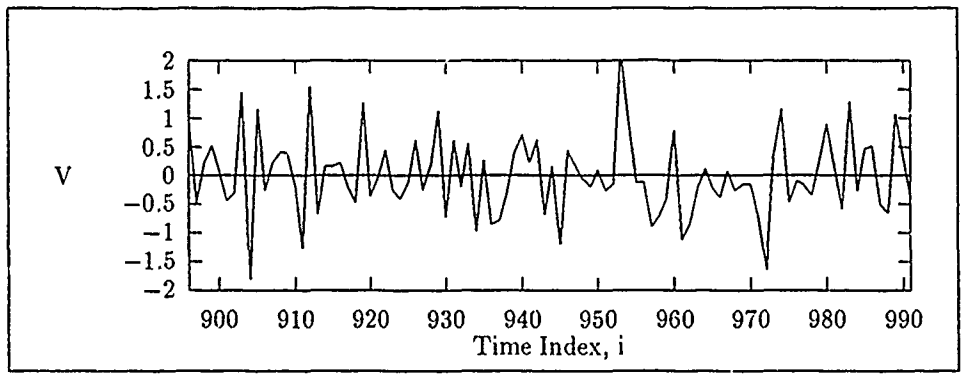

Figure 5.25. Signal Test 2: TDF filter output error for $3 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.1$.

Parameter	*Setting*
Block Size	16
Misadjustment	0.1
Datasize	992

Table 5.8. Signal Test 2: $4 - shift$ filter settings

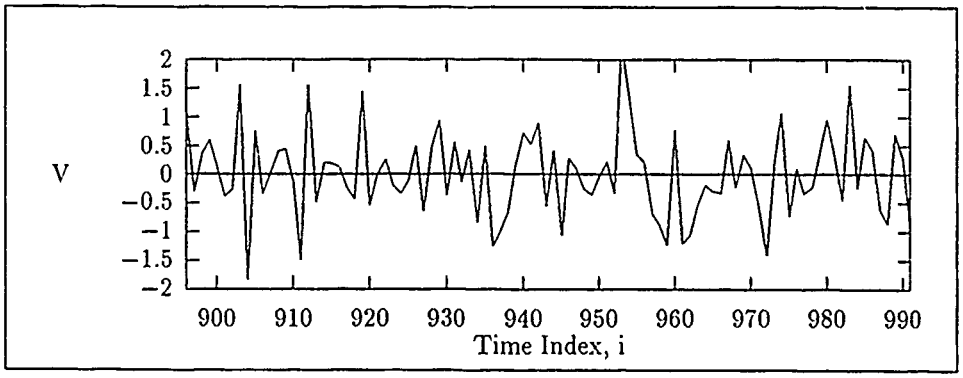

Figure 5.26. Signal Test 2: FDF filter output error for $3 - shift$ input. This is the FDF output error for the last 96 output samples using $M = 0.1$.

5.2.2.3 Noisy $4 - shift$ *Results.* The noisy $4 - shift$ signal pair is depicted in Figure 5.27. The filter configuration used for all 4 filters is specified in Table 5.8 while Figures 5.28, 5.29, 5.30, and 5.31 show the error for the last 96 output samples produced by each filter. As expected the Waslh-domain filters are unable to filter the $4 - shift$ signal (See Signal Test 1).

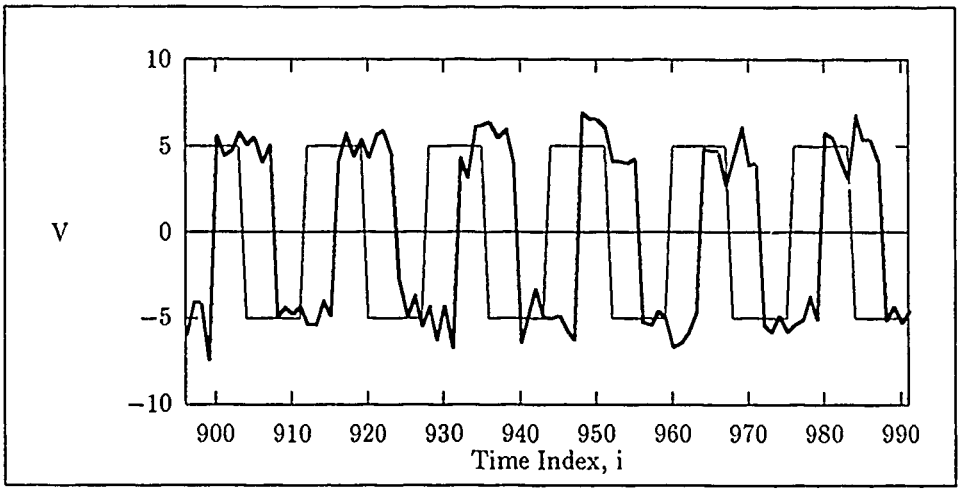

Figure 5.27. Signal Test 2: $4 - shift$ filter inputs. This is the last 96 samples of the noisy $4 - shift$ rectangular input and desired signal.

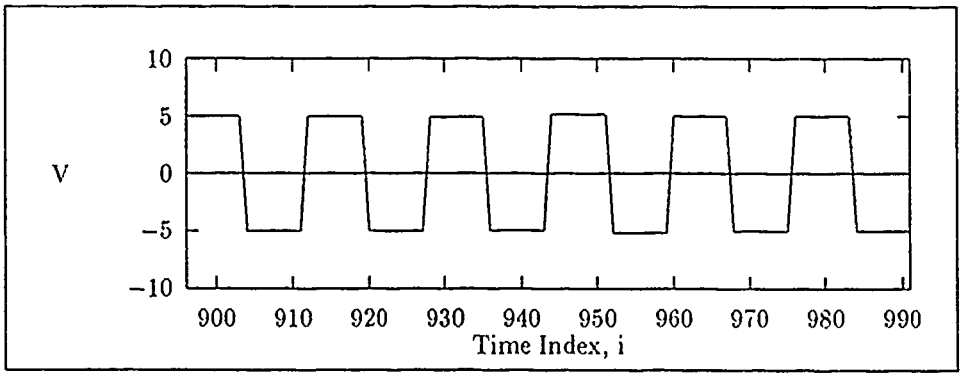

Figure 5.28. Signal Test 2: WDF1 filter output error for $4 - shift$ input. This is the WDF1 output error for the last 96 output samples using $M = 0.1$.

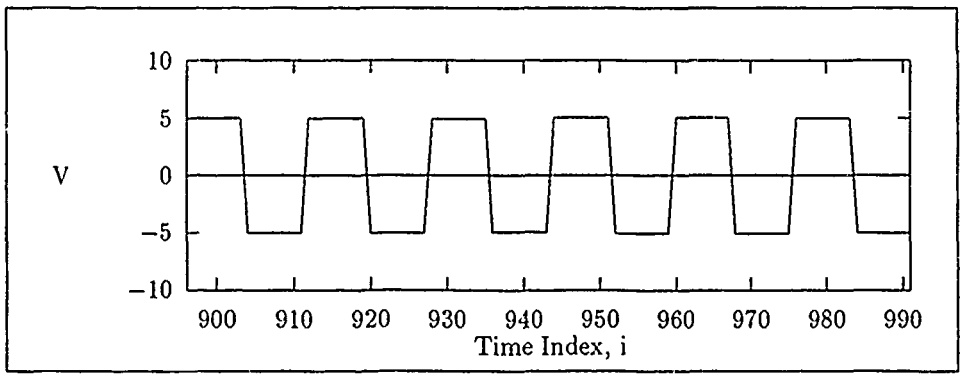

Figure 5.29. Signal Test 2: WDF2 filter output error for $4 - shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.1$.

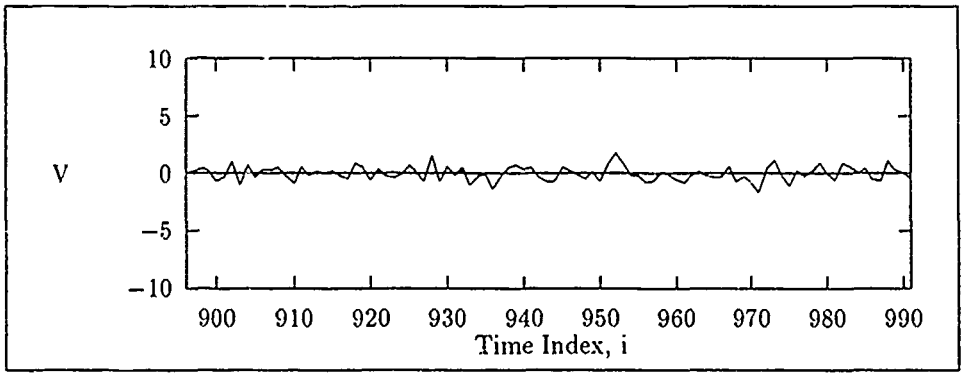

Figure 5.30. Signal Test 2: TDF filter output error for $4 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.1$.

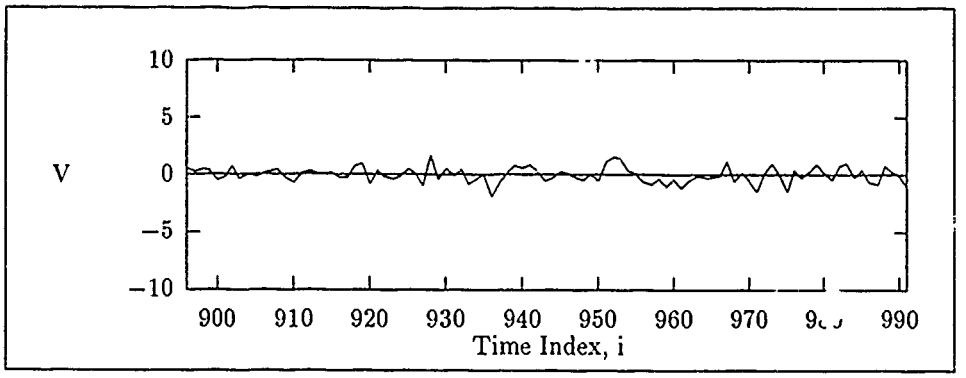

Figure 5.31. Signal Test 2: FDF filter output error for $4 - shift$ input. This is the FDF output error for the last 96 output samples using $M = 0.1$.

5.2.2.4 Signal Test 2 Analysis. For each of the signal sets in Signal Test 2, the three block processing filters performed optimally using a constant μ value. Using an independant μ value for each bin slowed the convergence speed of the filters relative to using a constant μ (See Signal Test 1). The FDF filter did not converge when an independant μ was used for each bin and all four filters converged more slowly as the reference shift was increased.

Error Performance. Considering the $0 - shift$ and $1 - shift$ cases, the Walsh-domain filter outputs were able to reasonably match the desired signal. The error associated with the WDF1 and WDF2 filter output was nominal and primarily consisted of a constant magnitude with occasional jumps corresponding with noise spikes in the input. The WDF2 response to the noise spikes was less pronounced than in the WDF1 case so that there was less variation in the error (Figures 5.19, 5.18). Conversely, the FDF and TDF filters were able to accomplish the necessary time shift, but unable to remove the noise from the input. The FDF filter was still converging at the end of the data set. This analysis is based on the filter configuration specified in Table 5.6.

The $2 - shift$ and $3 - shift$ inputs increase the output error for the Walsh-domain filters significantly as compared to the $0 - shift$ and $1 - shift$ inputs. The error signal is characteristically the same except larger in amplitude by a factor of 3 in the $2 - shift$ case and 4 in the $3 - shift$ case. The TDF and FDF filter error is characteristically the same and shows no appreciable change in amplitude. This analysis is based on the filter configuration specified in Table 5.7.

M	*Shift*	*Error Signal Power (Watts)*			
		TDF	FDF	WDF1	WDF2
0.05	0	$4.88E-01$	$7.48E-01$	$1.65E-01$	$6.01E-02$
0.05	1	$4.83E-01$	$7.31E-01$	$2.20E-01$	$8.46E-02$
0.05	2	$3.64E-01$	$3.77E-01$	$4.99E-01$	$1.86E-01$
0.1	3	$4.72E-01$	$5.37E-01$	1.49	$6.49E-01$
0.1	4	$3.85E-01$	$4.24E-01$	24.97	24.99

Table 5.9. Signal Test 2:Error signal power for the last 96 samples.

Table 5.9 contains the normalized error signal power for each filter for the last 96 error samples produced in each of the five noisy input cases. Comparatively, for the $0-shift$, $1-shift$, $2-shift$, and $3-shift$ cases, the WDF2 filter was better than the WDF1 filter in terms of minimizing the MSE. Compared to the TDF and FDF filters, the WDF2 filter performed better in terms of MSE for all four signal sets while the WDF1 filter performed better in terms of MSE for all but the three sample reference shift. The WDF1 and WDF2 filters were unable to filter the noisy $4-shift$ rectangular signal as Figures 5.29 and 5.28 show. The input spectrum in this case has no zero valued components due to the addition of the N(0,1) noise to the input.

The inability of the WDF1 and WDF2 filters to filter the noisy $4-shift$ rectangular signal arises from the fact that the spectrum components vary from block to block due to the noise. Variation in the ratio of the input spectrum values to the desired spectrum values disrupts the tap adaptation (See Sections 4.3 and 4.4). The particular bin of interest for WDF1 is bin 1, because the only nonzero desired spectral value is the sequency 1 term (See Section A.3). The particular bin of interest for WDF2 is bin 3, because the only nonzero desired spectral value is the sequency 3 term (See Section A.3). Figures 5.32 and 5.33 show the bin 1 tap and $Ratio_1(k)$ (See Equation 4.17) for WDF1. Figures 5.34 and 5.35 show the bin 3 tap and $Ratio_3(k)$ for WDF2.

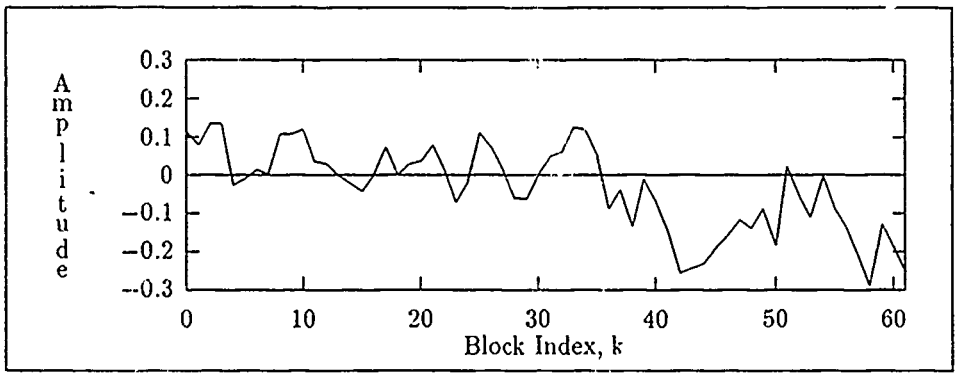

Figure 5.32. WDF1 Bin Tap $H_1(k)$. This figure depicts the WDF1 $H_1(k)$ adaptation track for the noisy $4 - shift$ rectangular signal.

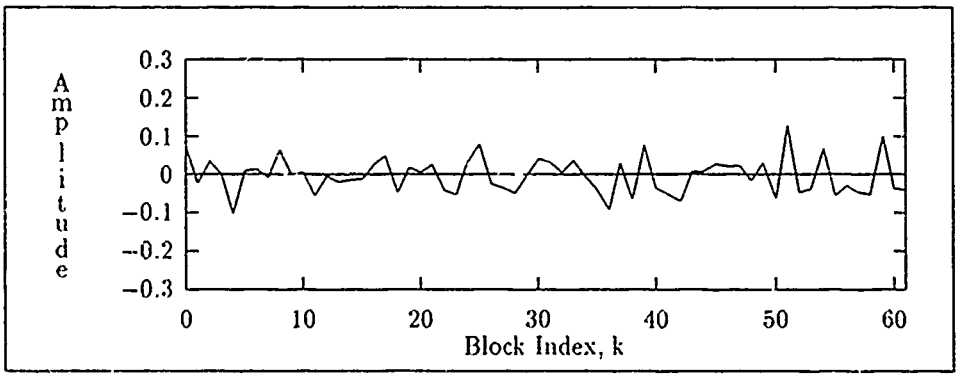

Figure 5.33. WDF1 $Ratio_1(k)$ This figure depicts the WDF1 spectral bin 1 input to desired ratio.

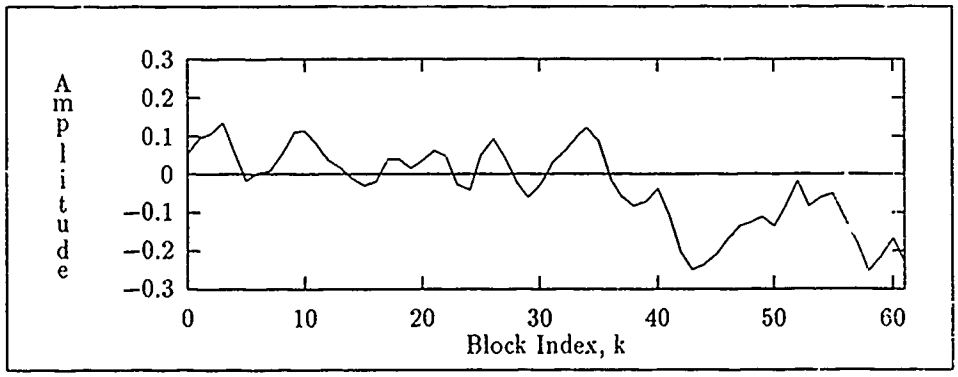

Figure 5.34. WDF2 Bin Tap $H_3(k)$. This figure depicts the WDF2 $H_3(k)$ adaptation track for the noisy $4 - shift$ rectangular signal.

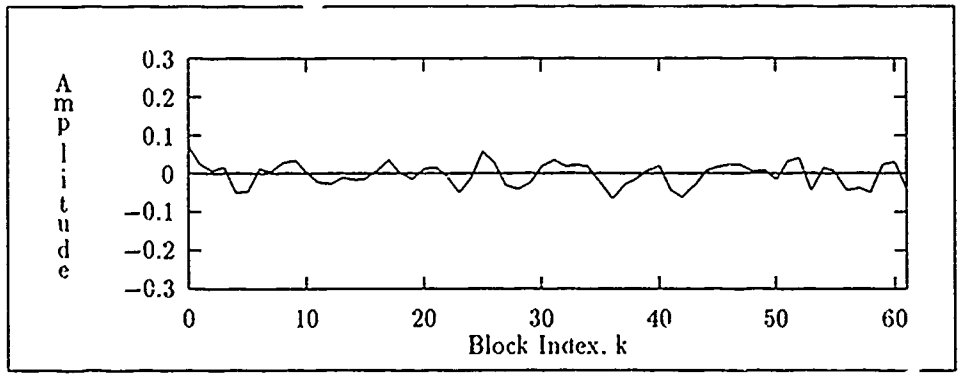

Figure 5.35. WDF2 $Ratio_3(k)$. This figure depicts the WDF2 spectral bin 3 input to desired ratio.

M	*Shift*	*Number of Weight Updates*			
		TDF	FDF	WDF1	WDF2
0.05	0	329	43	1	2
0.05	1	360	45	3	3
0.05	2	134	24	3	3
0.1	3	192	25	14	12
0.1	4	201	25	dnc	dnc

Table 5.10. Signal Test 2: Number of weight updates to converge

Convergence Speed Comparison. The results of signal test 2 in terms of number of weight updates required to converge are reflected in Table 5.10. The table shows in general that the blockprocessing filters require fewer weight updates for this test. Also, the Walsh-domain filters required the least number of weight updates, with the exception of the $4 - shift$ case. The WDF1 and WDF2 filters had similar performance for the $1 - shift$ and $2 - shift$ cases. The WDF1 filter was able to achieve convergence (i.e. 10% of the NMSE) after a single weight update whereas the WDF2 filter required two. The WDF2 filter outperformed the WDF1 filter on the $3 - shift$ case, only requiring twelve weight updates whereas the WDF1 filter required fourteen. The Walsh-domain filter entries for the $4 - shift$ input are indicated in the table as "dnc"; indicating "did not converge".

5.2.3 Signal Test 3. This test compares the WDF1 and WDF2 continuous signal filtering performance against the TDF and FDF filters. A simple sinusoidal signal is used as a continuous signal representative. The hypothesis is that the Walsh-domain filters will converge more slowly and produce more error filtering the sinusoidal signal than the FDF and TDF filters and that shifting the input will degrade the Walsh-filter performance. There are two attributes of the DWT which suggest this hypothesis:

1. The DWT of a continuous signal produces more spectral terms than the corresponding DFT spectrum does (See Section 2.1.2.2).
2. The DWT spectrum is not time-shift invariant (See Section 2.1.3).

The input signal used in the test is depicted in Figure 5.36 and serves as the input and desired signal in this test. A 992 sample datasize is used because it produces an integer number of 16-point blocks, as opposed to a datasize of 1000 which does not. The input

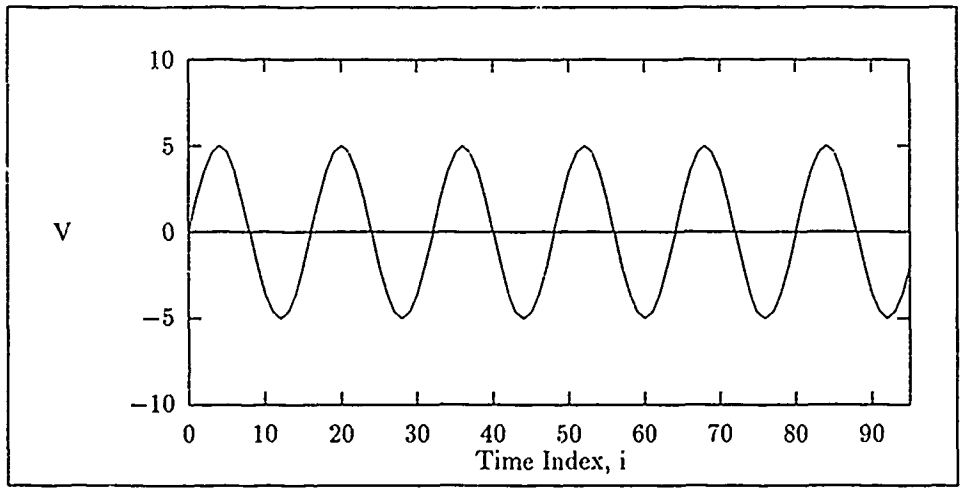

Figure 5.36. Signal Test 3 Sinusoidal Waveform.

signal was time-shifted to assess time-shift affects on filter error performance and convergence speed. An incremental shift of 1 sample is made on each filter run relative to the desired signal, with a maximum relative shift of 4 samples. For the remainder of this test, the input signal for an n-point relative shift is designated the $n - shift$ input. The results of the $4 - shift$ case are presented because it was representative of the results produced for all the shift cases. The purpose of this test is to evaluate the WDF1 and WDF2 sinusoidal signal filtering performance in relation to the TDF and FDF filters. The input signal used in the test is depicted in Figure 5.36. This signal functions as the input and desired signal. An incremental shift of 1 sample is made on each filter run relative to the desired signal, with a maximum relative shift of 4 samples. Only the results for the four sample reference shift are presented because it was representative of the results produced for all the shift cases.

5.2.3.1 Noiseless $4 - shift$ *Results.* The sinusoidal $4 - shift$ signal pair is depicted in Figure 5.37 and the filter configuration used for all 4 filters is specified in Table 5.11. Figures 5.38, 5.39, 5.40, and 5.41 show the error for the last 96 output samples produced by each filter.

Parameter	*Setting*
Block Size	16
Misadjustment	0.1
Datasize	992

Table 5.11. Signal Test 3: $4-shift$ filter settings.

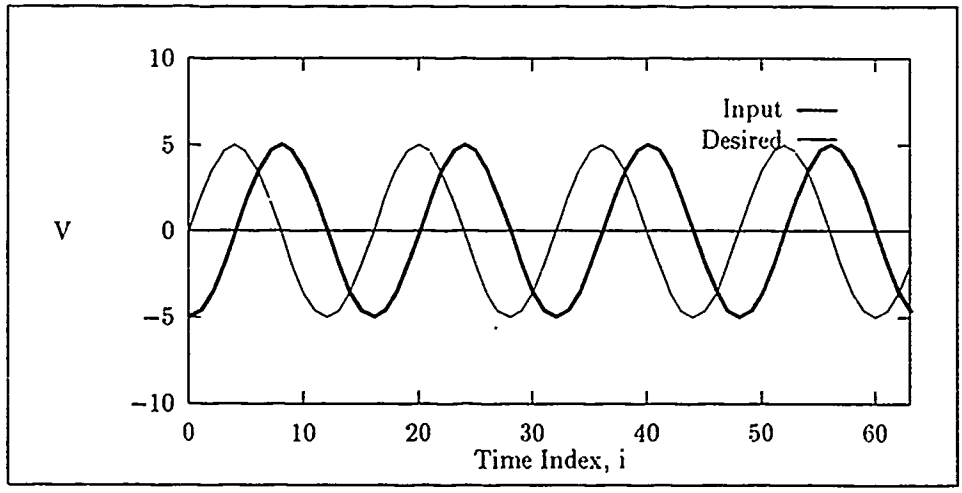

Figure 5.37. Signal Test 3: $4-shift$ filter inputs . This is the noiseless $4-shift$ sinusoidal input and desired signal.

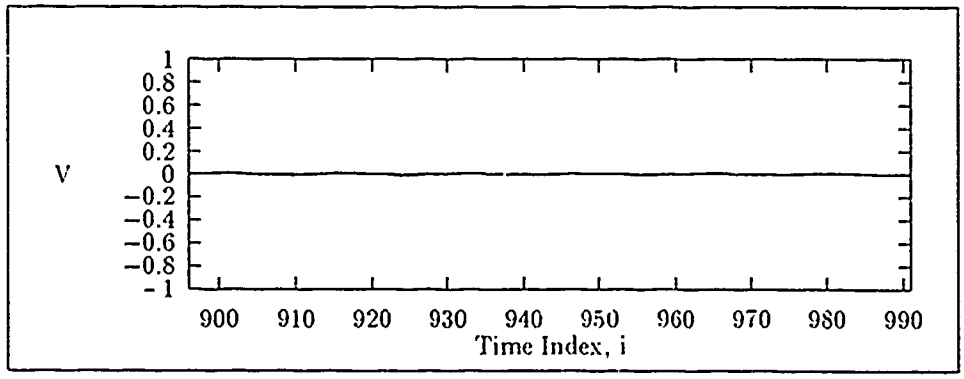

Figure 5.38. Signal Test 3: WDF1 filter output error for $4-shift$ input. This figure depicts the WDF1 output error for the last 96 output samples using $M = 0.1$.

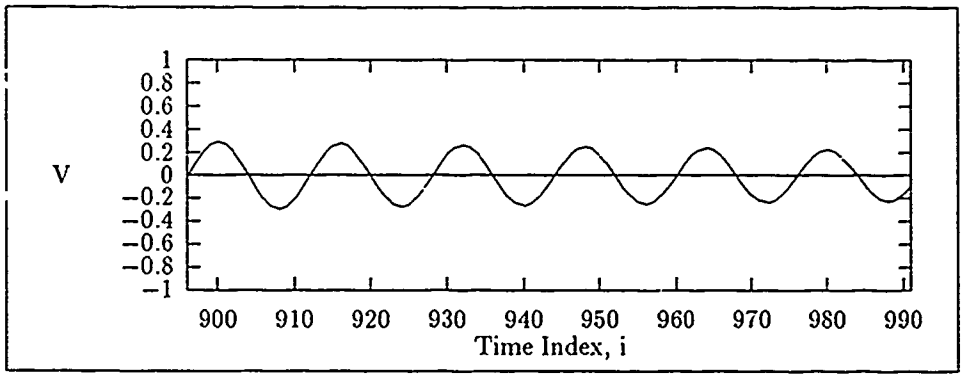

Figure 5.39. Signal Test 3: WDF2 filter output error for $4 - shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.1$.

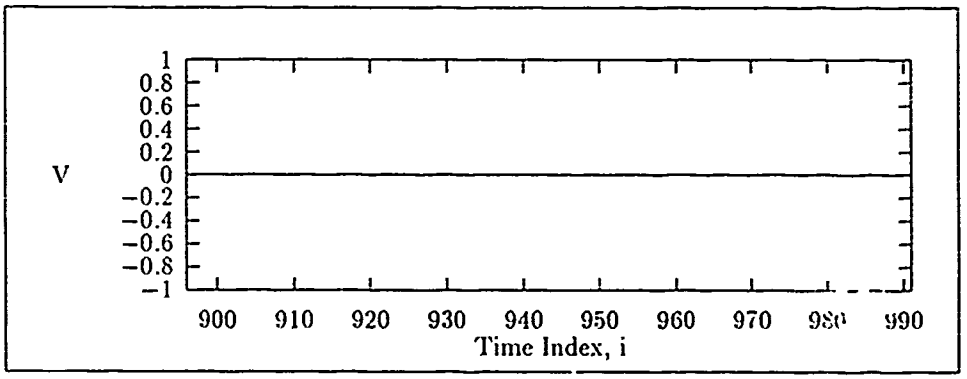

Figure 5.40. Signal Test 3: TDF filter output error for $4 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.1$.

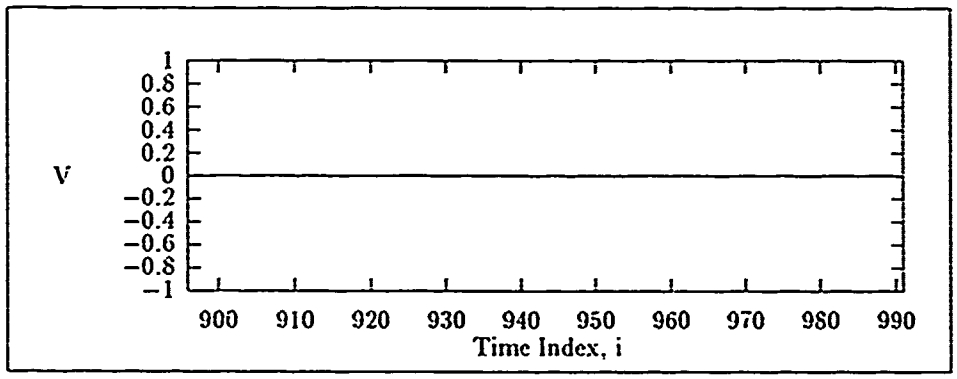

Figure 5.41. Signal Test 3: FDF filter output error for $4 - shift$ input. This is the FDF output error for the last 96 output samples using $M = 0.1$.

5.2.3.2 Signal Test 3 Analysis For the two Walsh-domain filters, an independant gain constant μ for each bin was optimum in terms of minimizing the output error while a constant μ value was optimum in terms of minimizing the error for the FDF filter in all cases. Figures 5.38 thru 5.41 show the output error over the last 96 samples for each filter. The optimum μ algorithm was used.

Error Performance. The results of test 3 in terms of error signal power, normalized to a one ohm resistor, for the last 96 error samples are contained in Table 5.12. Results for all sample shift cases produced indistinguishable squared error plots. The TDF and FDF filters produced a zero error output as Figures 5.40 and 5.41 show. The WDF1 filter error (Figure 5.38) was nominal compared to the WDF2 filter error (Figure 5.39).

Convergence Speed Comparison. Table 5.13 summarizes the results of signal Test 3 in terms of number of weight updates updates required to converge (i.e. reach 10% of normalized mean-square-error). Entries with an * designate results determined using an independant bin μ value. The table reflects that the blockprocessing filters, with the exception of WDF2 for shifts 3 and 4, required fewer weight updates to converge. The FDF filter was the quickest in terms of weight updates required; only needing two weight updates. The two Walsh-domain filters performed equally for the first two reference shifts; requiring five weight updates for the $1 - shift$ input and eight for the $2 - shift$ input. The WDF1 filter required less than half the number of weight updates required by the WDF2 filter for the 3 and $4 - shift$ signals.

M	*Shift*	*Error Signal Power (Watts)*			
		TDF	FDF	WDF1	WDF2
0.1	1	0	0	$5.90E-05$	$3.33E-02$
0.1	2	0	0	$5.90E-05$	$3.33E-02$
0.1	3	0	0	$5.90E-05$	$3.33E-02$
0.1	4	0	0	$5.90E-05$	$3.33-02$

Table 5.12. Signal Test 3:Error signal power for the last 96 samples.

M	*Shift*	*Number of Weight Updates*			
		TDF	FDF	WDF1	WDF2
0.1	1	21	2	5	5
0.1	2	21	2	8	8
0.1	3	21	2	10*	22*
0.1	4	15	2	10*	22*

Table 5.13. Signal Test 3: Number of weight updates to converge.

5.2.4 Signal Test 4 Signal test 4 is an extension of Signal Test 3 (See Section 5.2.3). This test evaluates the WDF1 and WDF2 noisy sinusoidal signal filtering performance in relation to the TDF and FDF filters. The WDF1 and WDF2 filter error should increase significantly relative to the Signal Test 3 results. This is based on the testing accomplished in Sections 4.3 and 4.4. The desired signal used in the test is the signal in Figure 5.36 and the input is the same signal with noise added. The filter input is simply a shifted version of the desired signal. The maximum shift was 4 samples. For the remainder of this test, the input signal for an n-point relative shift is designated the $n - shift$ input. Filter output results are presented for the $0 - shift$, $2 - shift$, and $4 - shift$ input cases, as they represent best, typical, and worst case.

5.2.4.1 Noisy $0 - shift$ *Results.* The sinusoidal $0 - shift$ signal pair is depicted in Figure 5.42 and Table 5.14 specifies the filter configuration. Figures 5.43, 5.44, 5.45, and 5.46 show the error for the last 96 output samples produced by each filter using the optimum μ calculation method.

Parameter	*Setting*
Block Size	16
Misadjustment	0.1
Datasize	992

Table 5.14. Signal Test 4: $0 - shift$ filter settings.

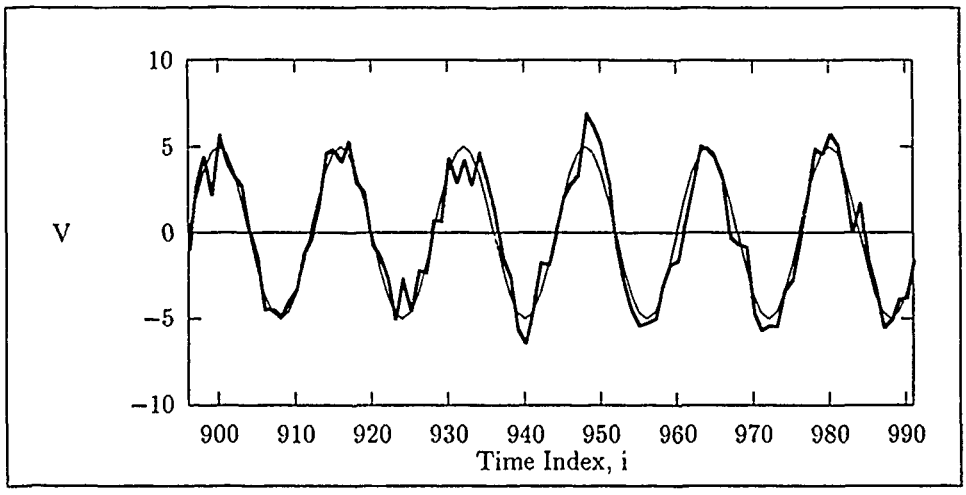

Figure 5.42. Signal Test 4: $0 - shift$ inputs. This is the last 96 samples of the noisy $0 - shift$ sinusoidal input and desired signal.

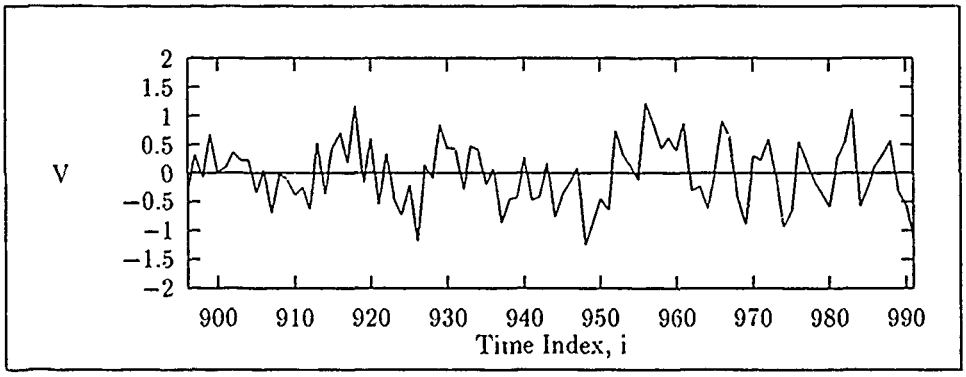

Figure 5.43. Signal Test 4: WDF1 filter output error for $0 - shift$ input. This is the WDF1 output error for the last 96 output samples using $M = 0.1$.

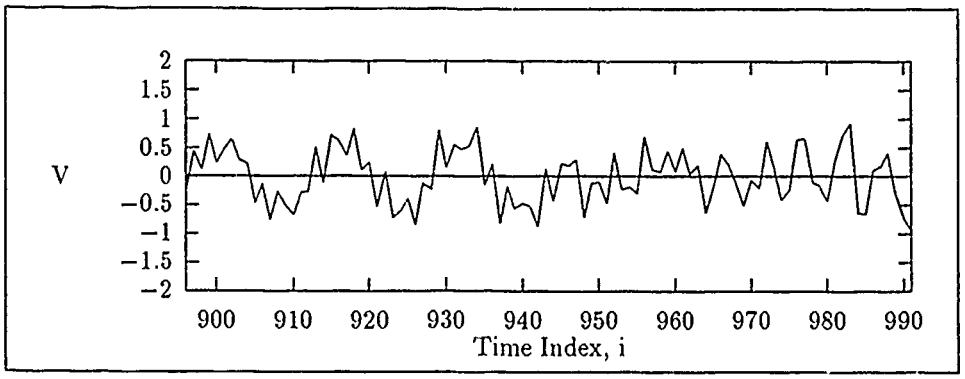

Figure 5.44. Signal Test 4: WDF2 filter output error for $0 - shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.1$.

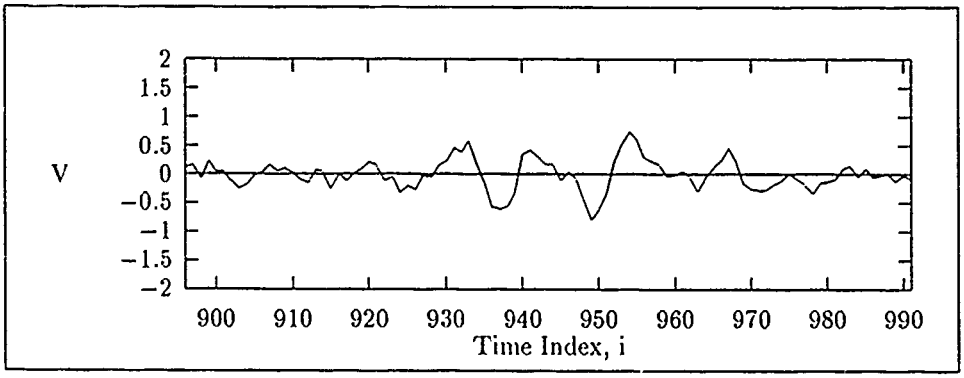

Figure 5.45. Signal Test 4: TDF filter output error for $0 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.1$.

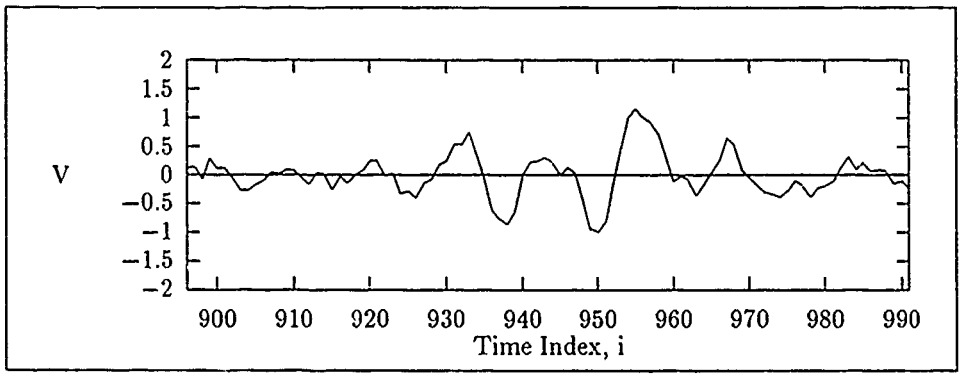

Figure 5.46. Signal Test 4: FDF filter output error for $0 - shift$ input. This is the FDF output error for the last 96 output samples using $M = 0.05$.

5.2.4.2 Noisy 2 – shift Results. The $2 - shift$ signal pair is depicted in Figure 5.47 and Table 5.15 specifies the filter configuration. Figures 5.48, 5.49, 5.50, and 5.51 show the error for the last 96 output samples produced by each filter using the optimum μ calculation method.

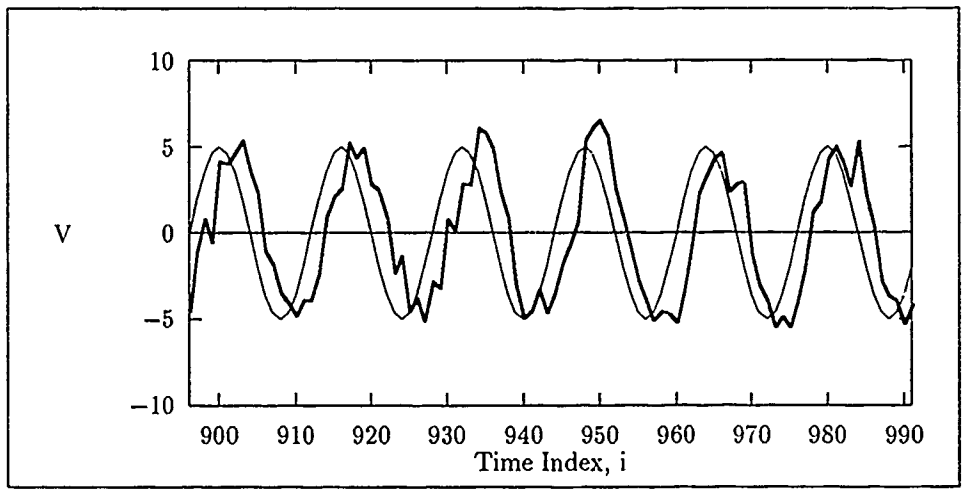

Figure 5.47. Signal Test 4: $2 - shift$ filter inputs. This is the last 96 samples of the noisy $2 - shift$ sinusoidal input and desired signal.

Parameter	*Setting*
Block Size	16
Misadjustment	0.1
Datasize	992

Table 5.15. Signal Test 4: $2 - shift$ input filter settings.

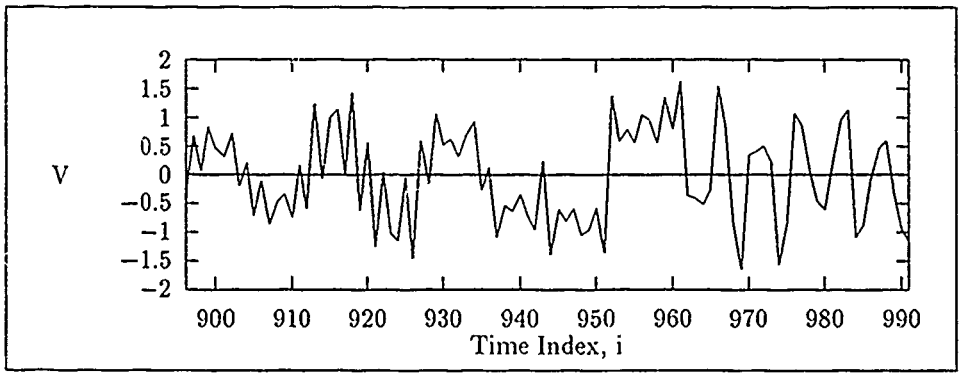

Figure 5.48. Signal Test 4: WDF1 filter output error for $2-shift$ input. This is the WDF1 output error for the last 96 output samples using $M = 0.1$.

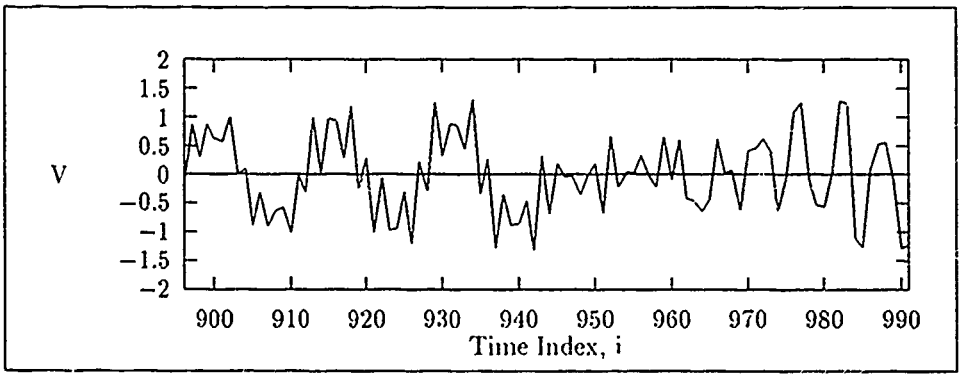

Figure 5.49. Signal Test 4: WDF2 filter output error for $2-shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.1$.

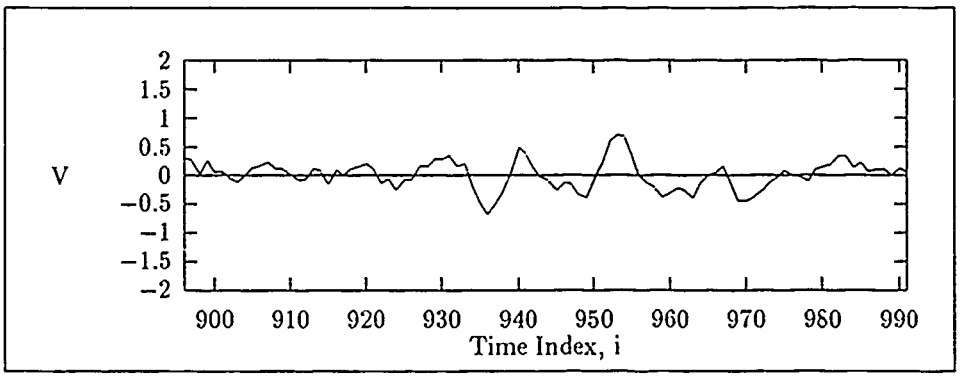

Figure 5.50. Signal Test 4: TDF filter output error for $2 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.1$.

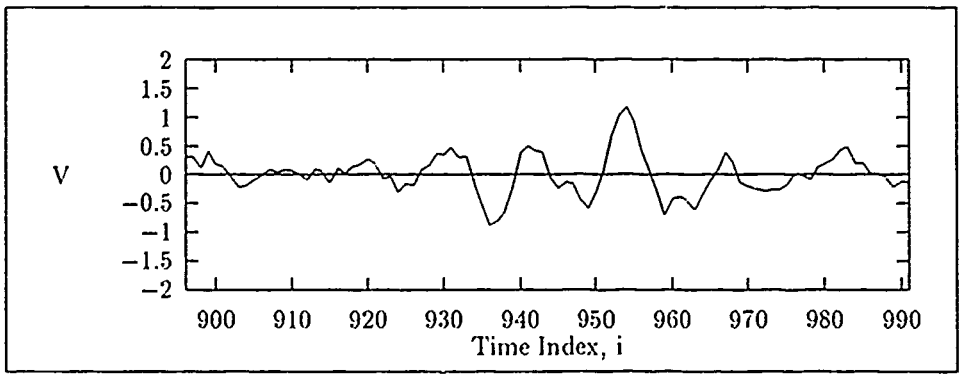

Figure 5.51. Signal Test 4: FDF filter output error for $2 - shift$ input. This is the FDF output error for the last 96 output samples using $M = 0.1$.

5.2.4.3 Noisy 4 – shift Results. The $4 - shift$ input pair is depicted in Figure 5.52. Table 5.16 specifies the filter configuration. Figures 5.53, 5.54, 5.55, and 5.56 show the error for the last 96 output samples produced by each filter using the optimum μ calculation method.

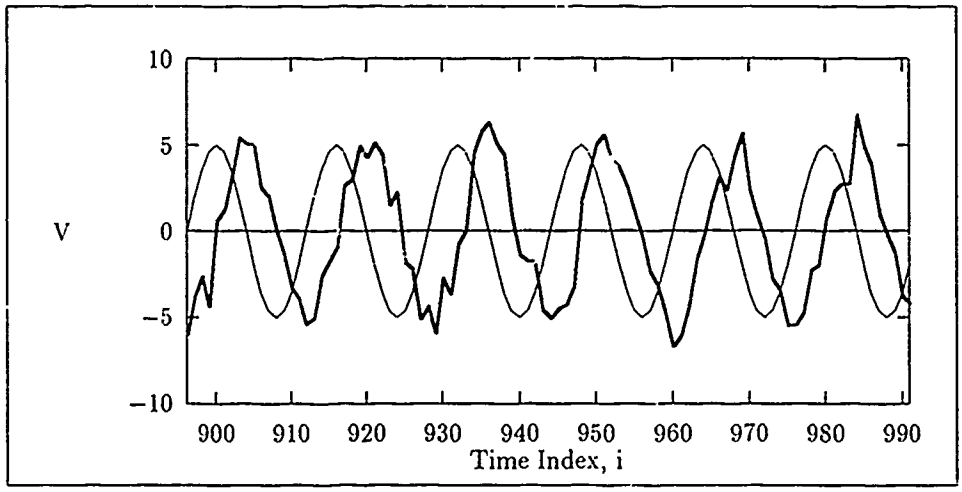

Figure 5.52. Signal Test 4: $4 - shift$ filter inputs. This is the last 96 samples of the noisy $4 - shift$ sinusoidal input and desired signal.

Parameter	*Setting*
Block Size	16
Misadjustment	0.1
Datasize	992

Table 5.16. Signal Test 4: $4 - shift$ filter settings.

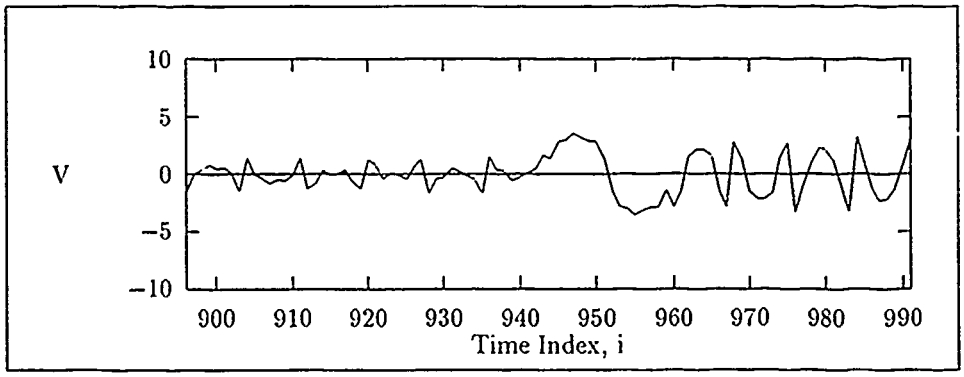

Figure 5.53. Signal Test 4: WDF1 filter output error for $4 - shift$ input. This is the WDF1 output error for the last 96 output samples using $M = 0.1$.

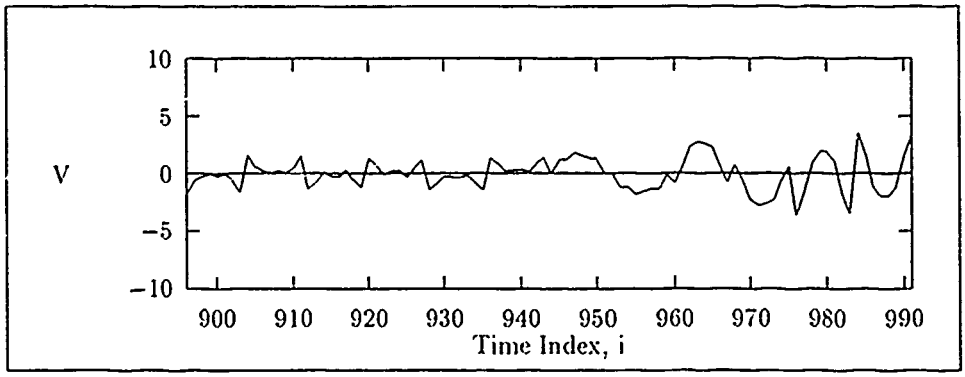

Figure 5.54. Signal Test 4: WDF2 filter output error for $4 - shift$ input. This is the WDF2 output error for the last 96 output samples using $M = 0.1$.

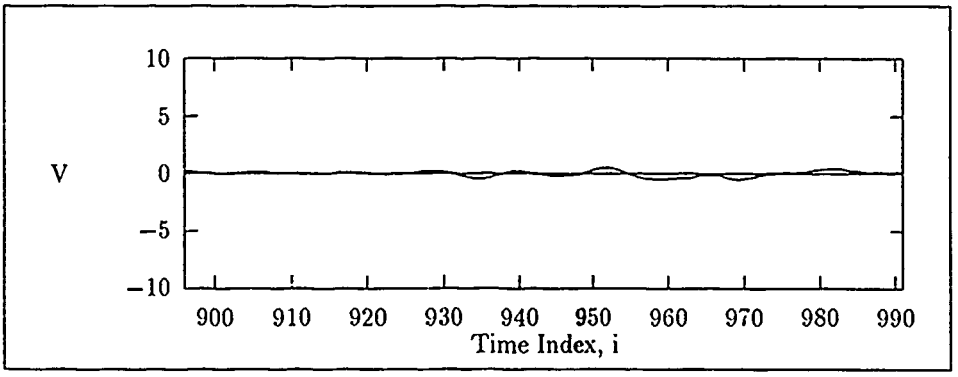

Figure 5.55. Signal Test 4: TDF filter output error for $4 - shift$ input. This is the TDF output error for the last 96 output samples using $M = 0.1$.

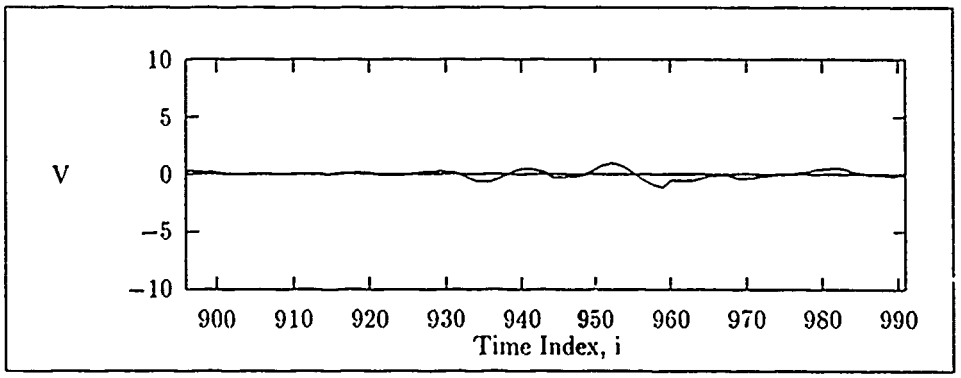

Figure 5.56. Signal Test 4: FDF filter output error for $4 - shift$ input. This is the FDF output error for the last 96 output samples using $M = 0.1$.

M	*Shift*	*Error Signal Power (Watts)*			
		TDF	FDF	WDF1	WDF2
0.1	0	$7.05E-02$	$1.49E-01$	$2.93E-01$	$2.18E-01$
0.1	1	$6.711E-02$	$1.38E-01$	$5.11E-01$	$2.67E-01$
0.1	2	$6.22E-02$	$1.26E-01$	$6.52E-01$	$4.67E-01$
0.1	3	$5.83E-02$	$1.20E-01$	3.19	2.13
0.1	4	$5.72-02$	$1.20E-01$	3.01	1.96

Table 5.17. Signal Test 4:Error signal power for the last 96 samples.

5.2.4.4 Signal Test 4 Analysis. An independant bin μ was optimum for WDF1 and WDF2 in minimizing the error for all reference shifts while the FDF filter did not converge for any reference shift using an independant bin μ.

Error Performance. Time shifting the input did not affect the output error for the TDF and FDF filters as clearly shown in Figures 5.46, 5.45, 5.50, 5.51, 5.55, and 5.56. The TDF error is, on average, approximately half that of the FDF filter. Figures 5.43, 5.44, 5.48, 5.49, 5.53, and 5.54 show that the output error for the Walsh-domain filters increases as the reference shift of the input increases while Table 5.17 contains the results of Signal Test 4 in terms of the error signal power, normalized to a one ohm resistor, for the last 96 error samples. The WDF2 output error is not as significant as the WDF1 filter and changes much less dramatically during the test than the WDF1 filter error. In both cases though, the error is much more significant than the FDF or TDF filter error.

The source of increasing error generated by the WDF1 and WDF2 filters is the progressive reduction in power of some of the input spectral components that correspond with the desired signal spectral components as the reference shift between the two signals increases (See Section A.3). For the noisy input, the signal to noise ratio of these bins is reduced which induces noise in the bin tap adaptation. Considering the noisy $2-shift$ sinusoid input, Figures 5.57 thru 5.60 show the resulting WDF1 input to desired bin ratios (Equation 4.17) for bins 9 and 13, and the resulting tap adaptation tracks for those bins. Figures 5.61 thru 5.64 show the same information concerning WDF2, which is using a 32-point transform of the input. In this case bins 19 and 27 are good examples of the adaptation noise created by bin ratio variation. Considering the $4-shift$ input, Figures 5.57 thru 5.60 address WDF1 bin numbers 9 and 13. Figures 5.69 thru 5.72, address WDF2 bin numbers 19 and 27.

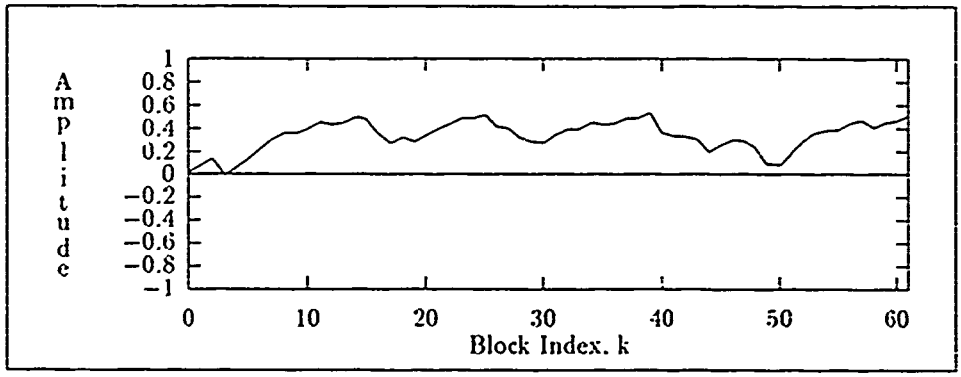

Figure 5.57. Signal Test 4: WDF1 filter tap $H_9(k)$ for the $2-shift$ input using independent bin μ and $M = 0.1$.

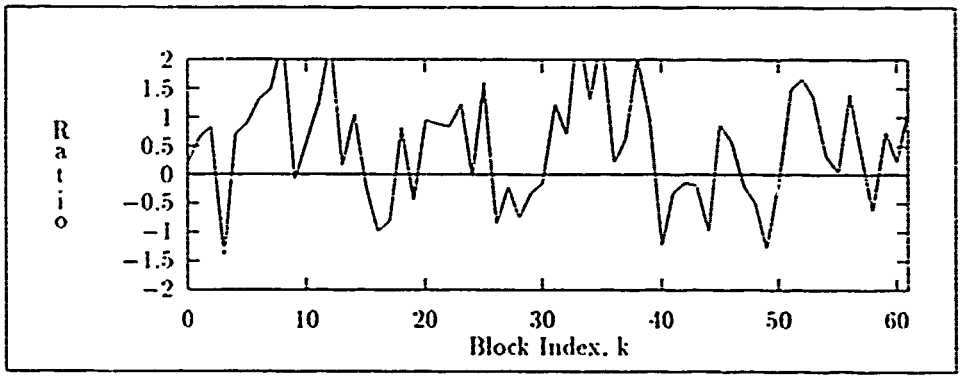

Figure 5.58. Signal Test 4.WDF1 $Ratio_9(k)$. This is the ratio of the $2-shift$ input 16-point DWT bin 9 and the desired signal 16-point DWT bin 9 versus k.

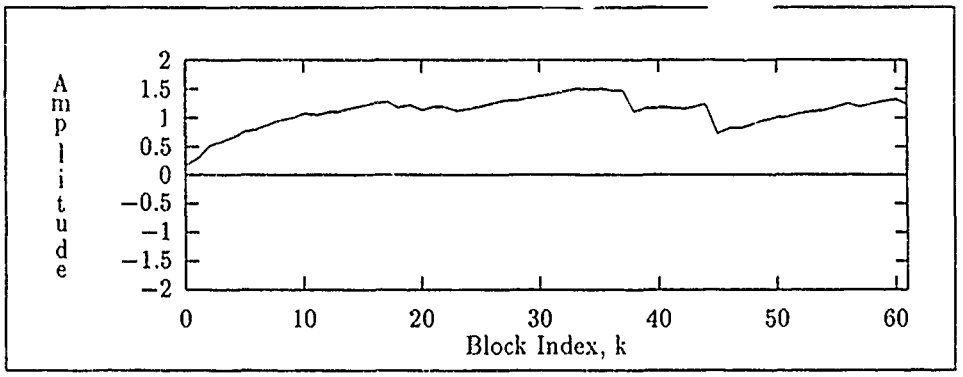

Figure 5.59. Signal Test 4: WDF1 filter tap $H_{13}(k)$ for the $2-shift$ input using independent bin μ and $M = 0.1$.

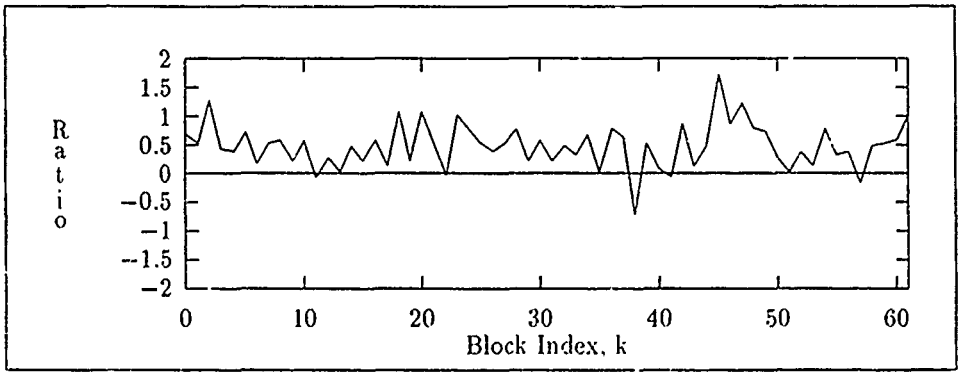

Figure 5.60. Signal Test 4:WDF1 $Ratio_{13}(k)$. This is the ratio of the $2 - shift$ input 16-point DWT bin 13 and the desired signal 16-point DWT bin 13 versus k.

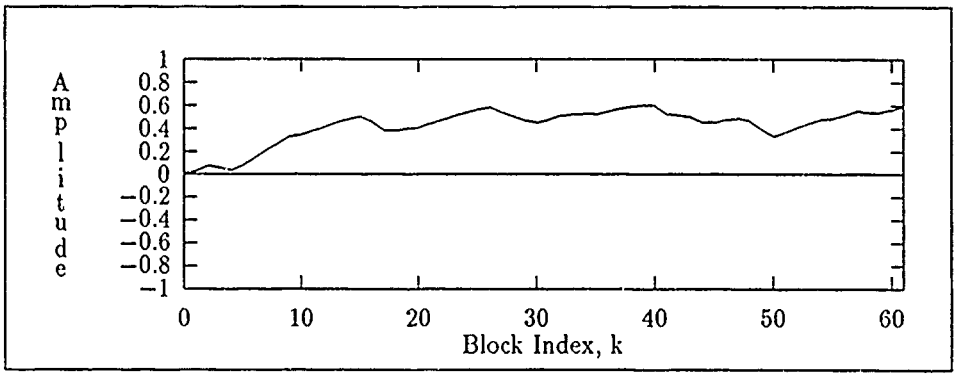

Figure 5.61. Signal Test 4: WDF2 filter tap $H_{19}(k)$ for the $2-shift$ input using independent bin μ and $M = 0.1$.

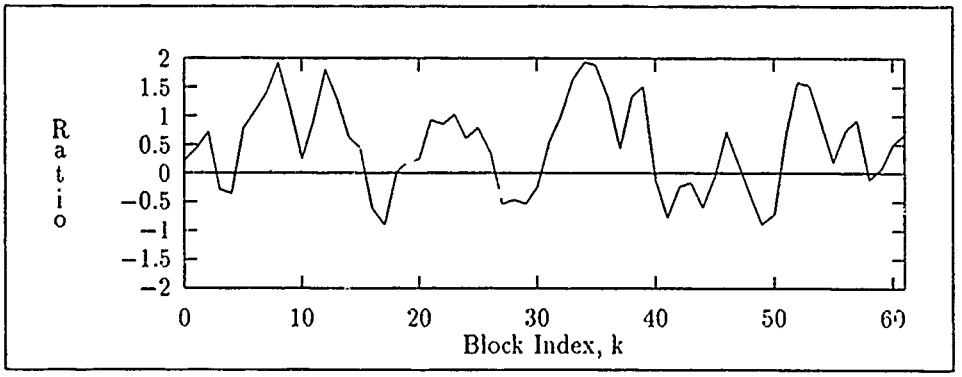

Figure 5.62. Signal Test 4:WDF2 $Ratio_{19}(k)$. This is the ratio of the $2 - shift$ input 32-point DWT bin 19 and the desired signal 32-point DWT bin 19 vers .s k.

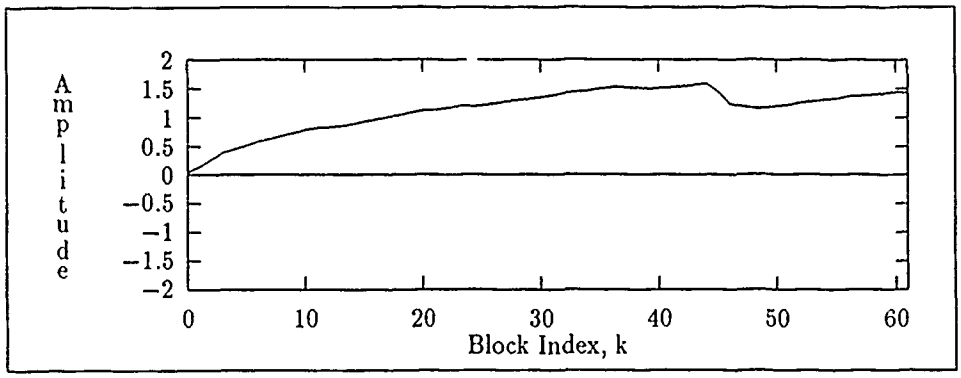

Figure 5.63. Signal Test 4: WDF2 filter tap $H_{27}(k)$ for the $2-shift$ input using independent bin μ and $M = 0.1$.

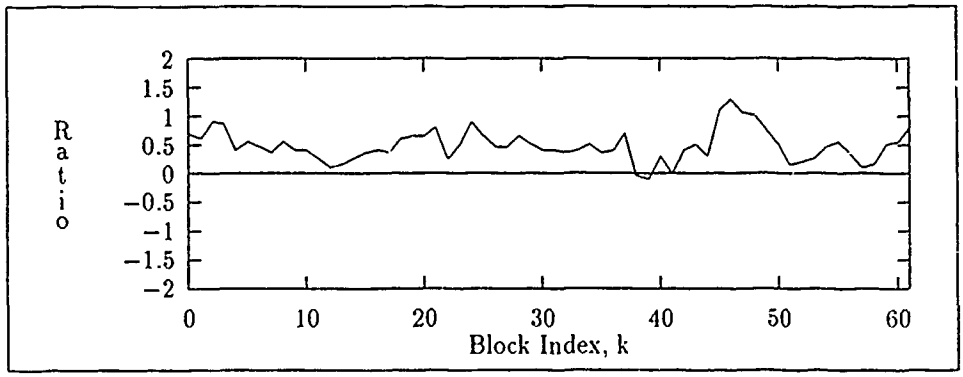

Figure 5.64. Signal Test 4:WDF2 $Ratio_{27}(k)$. This is the ratio of the $2 - shift$ input 32-point DWT bin 27 and the desired signal 32-point DWT bin 27 versus k.

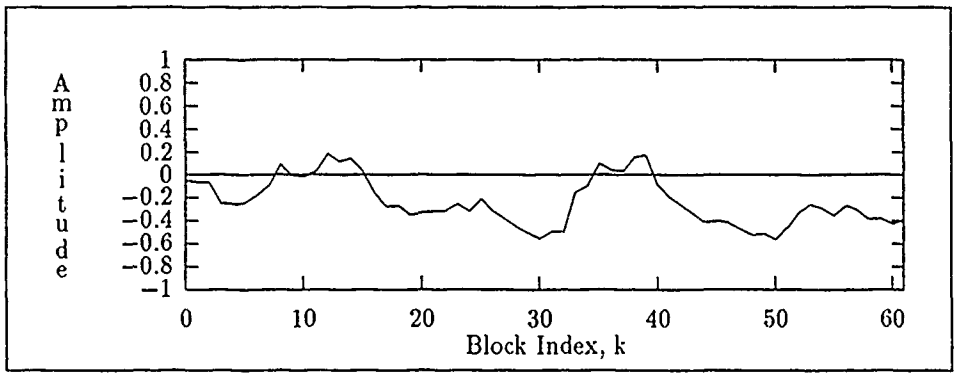

Figure 5.65. Signal Test 4: WDF1 filter tap $H_9(k)$ for the $4-shift$ input using independent bin μ and $M = 0.1$.

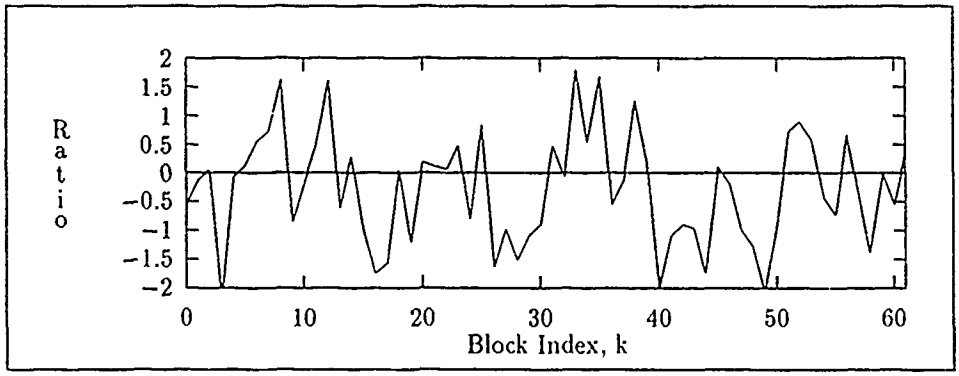

Figure 5.66. Signal Test 4:WDF1 $Ratio_9(k)$. This is the ratio of the $4-shift$ input 16-point DWT bin 9 and the desired signal 16-point DWT bin 9 versus k.

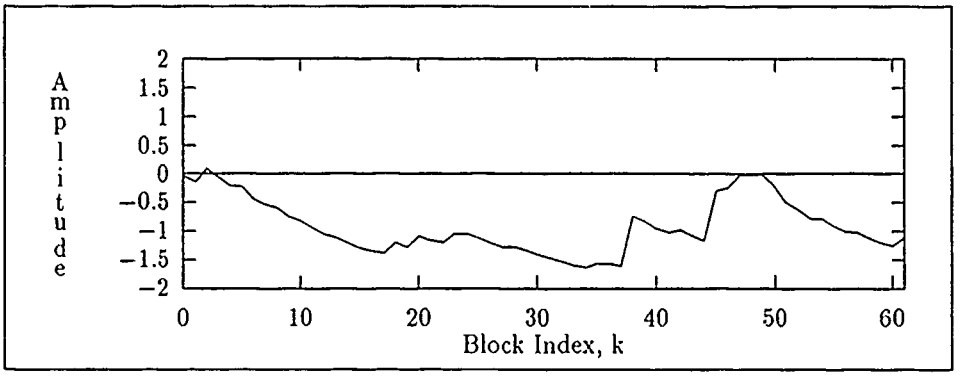

Figure 5.67. Signal Test 4: WDF1 filter tap $H_{13}(k)$ for the $4-shift$ input using independent bin μ and $M = 0.1$.

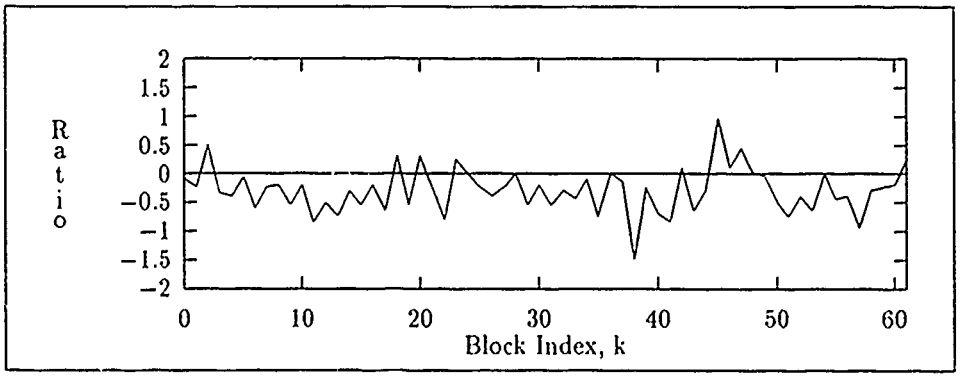

Figure 5.68. Signal Test 4:WDF1 $Ratio_{13}(k)$. This is the ratio of the $4 - shift$ input 16-point DWT bin 13 and the desired signal 16-point DWT bin 13 versus k.

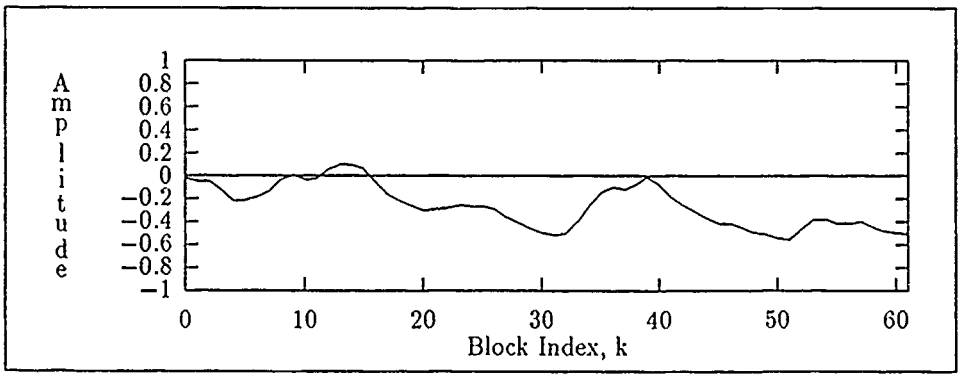

Figure 5.69. Signal Test 4: WDF2 filter tap $H_{19}(k)$ for the $4-shift$ input using independent bin μ and $M = 0.1$.

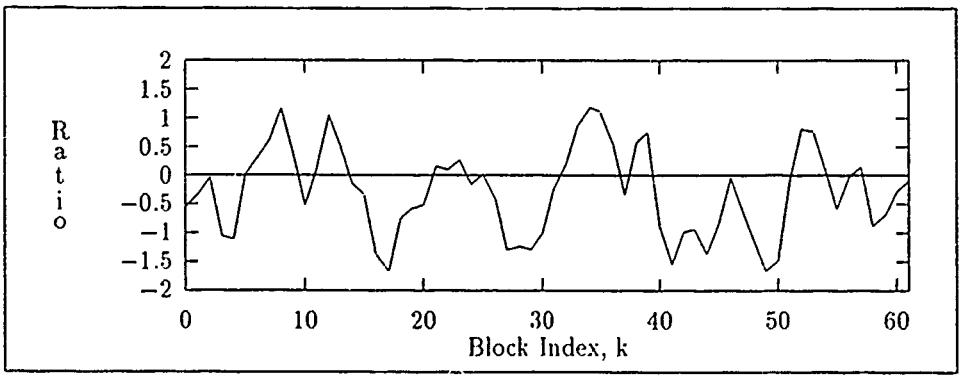

Figure 5.70. Signal Test 4:WDF2 $Ratio_{19}(k)$. This is the ratio of the $4 - shift$ input 32-point DWT bin 19 and the desired signal 32-point DWT bin 19 versus k.

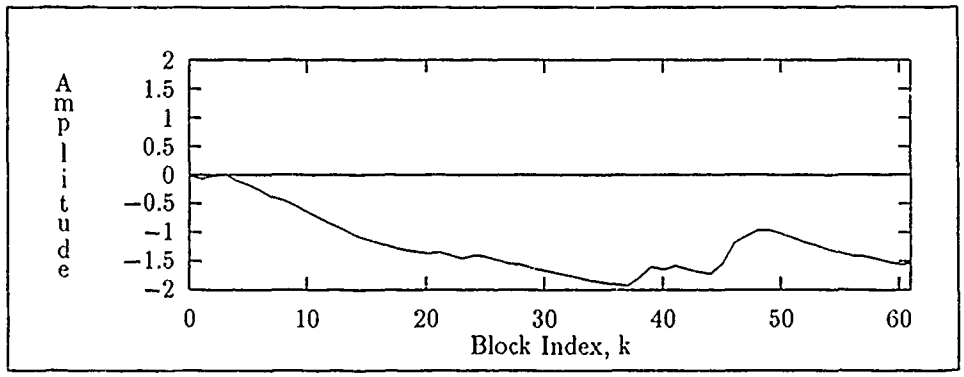

Figure 5.71. Signal Test 4: WDF2 filter tap $H_{27}(k)$ for the $4-shift$ input using independent bin μ and $M = 0.1$.

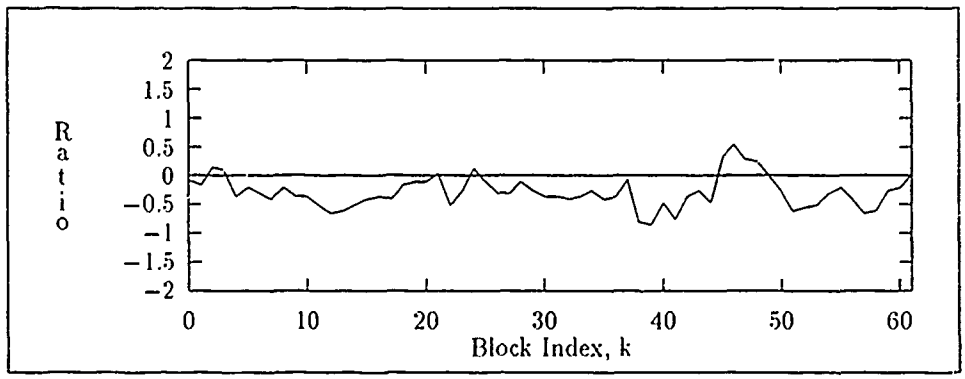

Figure 5.72. Signal Test 4:WDF2 $Ratio_{27}(k)$. This is the ratio of the $4 - shift$ input 32-point DWT bin 27 and the desired signal 32-point DWT bin 27 versus k.

M	*Shift*	*Number of Weight Updates*			
		TDF	FDF	WDF1	WDF2
0.1	0	21	1	9	9
0.1	1	21	2	5	6
0.1	2	22	2	10	10
0.1	3	22	2	61+	49*
0.1	4	21	2	61+	61+

Table 5.18. Signal Test 4: Number of weight updates to converge.

Convergence Speed Comparison. The convergence speed comparisons of signal test 4 are reflected in Table 5.18 which summarizes the number of weight updates required to converge. Entries with an * designate results determined using an independent bin μ value while an entry with a "+" indicates convergence was not achieved.

5.3 Filter Processing Speed Comparison

This section presents a processing speed performance comparison between the WDF1, WDF2, TDF, and FDF adaptive filters. Table 5.19 contains the time required for each filter to process a 1000 sample input sequence. The WDF1, WDF2, and FDF filters were configured to process blocksizes of $N = 8$ and $N = 4$ while the TDF filter was configured with 8 and 4 taps. Run-time for each filter began after all data constructs had been initialized, the data was loaded, and the gain constant was calculated. Also, all external and internal write statements were removed for the run. Run-time was stopped after all data had been processed.

The WDF1 and WDF2 filters used a Fast Walsh Transform (FWT) [7]. Computationally, the FWT requires $N \log_2 N$ summations for an N-point transform and no multiplies. The FDF filter used a Cooley-Tukey radix-2 FFT algorithm [6:435-436] that requires $2N \log_2 N - 4N$ real multiplies. The TDF filter requires $2N^2$ real multiplies to produce N output points (See Section 2.2.2.1).

As indicated in Table 5.19, the TDF filter recorded the fastest time for the $N = 4$ case and the WDF1 filter was the fastest for $N = 8$. Both WDF1 and WDF2 were much faster than FDF. The time required for the FWT and FFT routines to perform a 16-point transform is contained in Table 5.20, excluding set-up time. Clearly, the FWT is much faster

than the FFT. The FDF filter requires 5 FFTs per block to produce N output points (See Section 2.2.2.2). The FWT's faster processing time and the fact that the Walsh-domain filters are using only three tranforms per block (See Sections 3.1 and 3.2) accounts for the faster processing times relative to FDF.

N	*Processing Time in Seconds*			
	TDF	FDF	WDF1	WDF2
4	4.22	52.68	6.87	9.89
8	7.85	60.80	7.74	10.00

Table 5.19. Processing Time: Time required to process 1000 data samples. For TDF N indicates number of taps; N indicates blocksize otherwise.

Transform	*Processing Time in Seconds*
FWT	0.05
FFT	0.11

Table 5.20. Time required to perform a 16-point transform.

5.4 Summary.

This purpose of this chapter was to establish a performance measure for WDF1 and WDF2, relative to each other and the TDF and FDF filters. In terms of convergence speed and error performance, the Walsh-domain filters' performance degraded as the reference shift was increased in all 4 signal tests.

Since numerous observations were made in the course of testing error performance of the filters in this chapter it would be wise to readdress each of them at this time. The major error performance results from this chapter are as follows:

1. In Signal Test 3 (See Section 5.2.3) all 4 filters produced error results that were invariant to the shift.
2. The $4 - shift$ rectangular signal in Signal Test 1 (Section 5.2.1) and Signal Test 2 (Section 5.2.2) produced the worst performance for the WDF1 and WDF2 filters.

- In Signal Test 1 the $4-shift$ input results in a single sequency component DWT for the desired and a different single sequency DWT for the input. Since the signals are periodic and the block size matches the period, both DWTs are constant from block to block. With single nonzero noncoincident components, the input cannot be filtered to achieve the desired signal.
- In Signal Test 2, the input DWT component position corresponding with the desired DWT nonzero component is due to noise only. In this case, the variation of the input bin of interest is such that the bin tap cannot converge. This problem also arises for the WDF1 and WDF2 filters in Signal Test 4 (Section 5.2.4).

3. The $3-shift$ and $4-shift$ noiseless sinusoidal spectrums result in low power components that correspond with higher sequency components in the desired spectrum. Adding noise to the input results in a low signal-to-noise-ratio (SNR) for these low power bins. In filtering this noisy spectrum to achieve the desired spectrum, the low SNR bin tap gradient follows the continuously changing relationship between the constant valued desired bins and the noisy input bins. Thus, the bin tap adaptation tracks are correspondingly noisy which limits the filtering abilty of the WDF1 and WDF2 filters.
4. In general, the WDF2 filter error signal performance was better for the noisy input cases, relative to the WDF1 filter.
5. The WDF1 filter error performance was better for the noiseless input cases, relative to the WDF2 filter.

In this thesis, convergence speed was measured in terms of the number of weight updates required to achieve 10% NMSE (See Section 5.1). Using this criteria, the Walsh-domain filters were faster than the TDF and FDF filters when tested using rectangular signals. The WDF1 filter was better than WDF2 in Signal Test 1 while WDF2 was better in Signal Test 2. The FDF filter required the least number of weight-updates for the sinusoidal input used in Signal Test 3 and 4. Comparing the two Walsh-domain filters; WDF1 required as many or fewer weight-updates as WDF2 did for the noiseless sinusoidal input in Signal Test 3. WDF2 required as many or fewer weight-updates as WDF1 did for the noisy sinusoidal input in Signal Test 4.

A processing speed comparison (See Section 5.3) showed the Walsh-domain filters to be at least four times as fast as FDF for $N = 8$. The WDF1 filter was the fastest of the

blockprocessing filters used in this thesis followed by the WDF2 and FDF filters, in that order. This is based on the FDF filter using a radix-2 FFT and the Walsh-domain filters using a Fast Walsh Transform. The TDF filter was slightly slower than the WDF1 filter for a blocksize of $N = 8$. The next chapter presents the conclusions and recommendations for this research effort.

VI. Conclusions and Recommendations

6.1 Conclusions

This thesis investigated the development of two Walsh-domain adaptive filters. The first, WDF1, was implemented using a frequency-domain circular convolution design while the second, WDF2, was implemented using a modified Fast LMS design. A time-domain adaptive filter (TDF) and the Fast LMS filter (FDF) were also implemented in software and used for comparison. Rectangular and sinusoidal test signals were used. Shifted noisy and noiseless versions of both were used to conduct error performance, convergence speed, and processing speed performance comparisons for the TDF, FDF, WDF1, and WDF2 filters. The following subsections address the performance comparison conclusions.

6.1.1 Error Performance Conclusions. Based on the last 96 error samples produced by each filter, the error performance conclusions from this research effort are the following:

- The Walsh-domain filters are better than the TDF and FDF filters in terms of mean-square-error (MSE) filtering discontinuous input signals, while the TDF and FDF filters are better for continuous signals.
- Input shifts cause WDF1 and WDF2 error performance to degrade for noisy and noiseless discontinuous signals.
- The WDF1 filter is better than the WDF2 filter for noiseless signals while the WDF2 filter is better than the WDF1 filter for noisy signals.
- A single μ is better than separate μ's for Walsh-domain filters filtering rectangular input signals while a separate μ is better for sinusoidal input signals.
- Shifts in the input data do not degrade WDF1 and WDF2 performance when filtering noiseless sinusoidal signals. That is not the case for noisy signals.
- Shifts in the input data do not affect TDF and FDF MSE performance.
- The FDF filter produces less error using a single bin μ for simple sinusoids and rectangular signals, relative to the results obtained using a separate bin μ.

6.1.2 Convergence Speed Performance Conclusions. The convergence speed criteria was the number of weight updates required to achieve 10% Normalized Mean Square Error (NMSE) where normalization was with respect to the desired signal power. For the noisy input signals, MSE learning curves were generated from 100 data files using Additive White Gaussian Noise (AWGN). Using this criteria it was concluded that:

- The Walsh-domain filters converged more slowly when filtering sinusoidal input signals as opposed to rectangular input signals.
- The Walsh-domain filters converged more quickly for noisy and noiseless rectangular input signals and more slowly when filtering noisy and noiseless sinusoidal signals, relative to the FDF filter.
- The TDF filter converged more slowly than the FDF, WDF1, and WDF2 filters when filtering noisy and noiseless rectangular signals.
- Beyond a two sample shift of the input, the Walsh-domain filters converge more quickly using a separate bin μ when filtering noisy and noiseless sinusoidal signals.
- Beyond a two sample shift of the input, the TDF converges more quickly than WDF2 for the noiseless sinusoidal signal and more quickly than WDF1 and WDF2 for noisy sinusoidal signals.
- Convergence speed was the same for WDF2 and WDF1 for noisy and noiseless rectangular signals.
- The WDF2 sinusoidal input signal convergence speed for the noiseless input degrades more quickly as the input shift increases and degrades at the same rate for noisy input. relative to the WDF1 sinusoidal input signal convergence speed performance.
- The Walsh-domain filters converge more quickly filtering a rectangular input signal using a single bin μ. relative to the results obtained using a separate bin μ.
- Data shifting degraded the convergence speed of the Walsh-domain filters.
- The convergence speed of the TDF and FDF filters was independant of the input shift.

6.1.3 Processing Speed Performance Conclusions. The FDF filter was implemented with a radix-2 FFT while the Walsh-domain filters used a Fast Walsh Transform (FWT). The FDF, WDF1, and WDF2 filters were configured to process 8-point and 4-point block sizes and the TDF filter was configured with 8 and 4 taps. The datasize was 1000 samples

and the run time started after all data constructs had been initialized, the data was loaded, and the gain constant was calculated. Based on this configuration, the processing speed performance conclusions for this research effort are the following:

- The TDF filter processed the 1000 samples more quickly than the FDF, WDF1, and WDF2 filters for small N.
- The WDF1 filter processed the 1000 data samples more quickly than the FDF and WDF1 filters.
- The WDF1 filter is faster than the WDF2 filter.
- The FDF filter processes the 1000 samples more slowly, relative to the TDF, WDF2, and WDF1 filters.

6.1.4 Filtering Limitations.

- A WDF1 and WDF2 filtering limitation exists due to the time-shift variant nature of the DWT. This is based on the fact that a signal data shift can result in a DWT that has different sequency terms than the DWT of an unshifted version of the signal. The $4 - shift$ input signal in Signal Test 2 was an occurrence of this.
- The quality of the convergence characteristics for WDF1 and WDF2 is inversely proportional to the variance of the input-to-desired sequency ratios.

6.2 Subjective Ranking

The TDF, FDF, WDF1, and WDF2 filters can be subjectively ranked using the results of Chapter 4. Table 6.1 provides subjective rankings for the filters when the noiseless and noisy, sinusoid and rectangular inputs were used. A rank of 1 indicates best. Overall performance was considered for each input signal. The $4 - shift$ rectangular input was not included in the rankings presented because it was considered an exception.

6.3 Recommendations

There are several recommendations which might make a reasonable thesis topic or serve as topics within related research.

1. The WDF2 filter implemented in this thesis used a 50% overlap to create the kth block input vector. One might further investigate what effects the choice of another overlap percentage would have on the filtering performance of the filter.

2. The Walsh-domain filters were implemented using a modified form of the complex LMS algorithm. One could also implement the filters using a modified form of the Leakage LMS algorithm to investigate whether this would improve the general convergence characteristics of the Walsh-domain bin taps.

3. Finally, this thesis used a simple representative of a continuous and discontinuous gnal to test the Walsh-domain filters. To further evaluate the filters, more complex representatives of each type of signal could be used.

Criteria	*Sine Input (Noiseless/Noisy)*				*Rect Input (Noiseless/Noisy)*			
	TDF	FDF	WDF1	WDF2	TDF	FDF	WDF1	WDF2
Process Time	2/2	4/4	1/1	3/3	2/2	4/4	1/1	3/3
Convergence Speed	4/2	1/1	2/3	3/3	4/4	3/3	1/1	2/2
Error	1/1	1/2	2/4	3/3	2/2	3/3	1/4	1/1

Table 6.1. Subjective Ranking for Noiseless/Noisy Input.

Appendix A. *Discrete Walsh Transform*

This Appendix presents examples of Discrete Walsh Functions and the Discrete Walsh Transform (DWT) and its properties.

A.1 Discrete Walsh Functions

For a series of N terms, the discrete Walsh functions can be specified as [1:59]

$$WAL(n,i) = \prod_{r=0}^{p-1} (-1)^{n_{p-1-r}(i_r + i_{r+1})} \tag{A.1}$$

$$i, n = 0, 1, 2, \ldots, N-1 \tag{A.2}$$

$$r = 0, 1, 2, \ldots, p-1 \tag{A.3}$$

where $N = 2^p$ defines p. The indexes i, n are expressed in terms of their binary digits such that

$$\begin{aligned} i &= (i_p, i_{p-1}, \ldots, i_1, i_0)_2 \\ n &= (n_p, n_{p-1}, \ldots, n_1, n_0)_2 \end{aligned} \tag{A.4}$$

An example calculation for the $N = 4$ series term $WAL(3,1)$ using Equations A.1 and A.5 is presented. In evaluating $WAL(3,1)$ we start with

$$N = 4 = 2^2$$

which means there are $p = 2$ product terms, so that

$$WAL(3,1) = \prod_{r=0}^{1} (-1)^{n_{1-r}(i_r + i_{r+1})}$$

and

$$\begin{aligned} i &= (0\ 1)_2 \\ n &= (1\ 1)_2 \end{aligned}$$

The first product term ($r = 0$) then, is

$$\begin{aligned} &= (-1)^{n_1(i_0+i_1)} \\ &= (-1)^{1\cdot(0+1)} \\ &= -1 \end{aligned}$$

The second product term ($r = 1$) is

$$\begin{aligned} &= (-1)^{n_0(i_1)} \\ &= (-1)^{1\cdot 0} \\ &= 1 \end{aligned}$$

$WAL(3,1)$ can now be calculated by multiplying the first and second product terms together, so that

$$\begin{aligned} WAL(3,1) &= -1\cdot 1 \\ &= -1 \end{aligned}$$

Figure A.1 shows the $N = 8$ series Discrete Walsh functions.

A.2 Dyadic Convolution

Dyadic convolution is defined as [1:100]

$$z_\tau = 1/N \sum_{i=0}^{N-1} x_i y_{\tau \oplus i} \tag{A.5}$$

and using the Discrete Walsh Transform (See Section 2.1.2)

$$z_\tau = \sum_{n=0}^{N-1} X_n Y_n WAL(n,\tau) \tag{A.6}$$

Two $N = 4$ point time-domain sequences x_i and y_i are given by

$$x_i = \{1\ 5\ 3\ 8\}$$

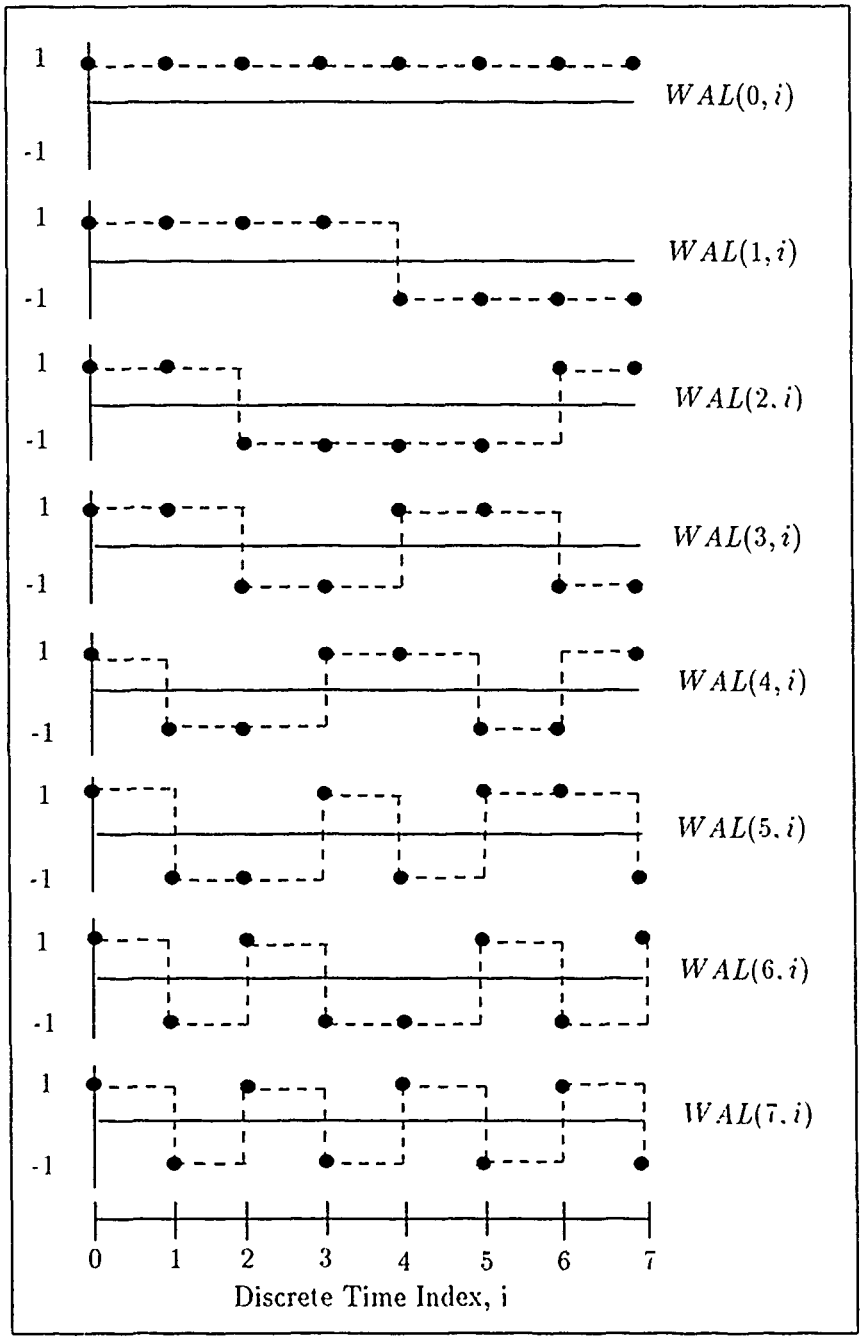

Figure ... Discrete Walsh Functions for $N = 8$, in sequency order.

$$y_i = \{2\ 7\ 4\ 1\}$$

where the x_0 and y_0 terms are the 1 and 2 terms respectively. Using Equation A.5, for $\tau = 0$ we have

$$\begin{aligned} z_0 &= 1/4 \sum_{i=0}^{3} x_i y_{0\oplus i} \\ &= 1/4[x_0y_0 + x_1y_1 + x_2y_2 + x_3y_3] \\ &= 1/4[1{\cdot}2 + 5{\cdot}7 + 3{\cdot}4 + 8{\cdot}1] \\ &= 14.25 \end{aligned}$$

For $\tau = 1$ the result is

$$\begin{aligned} z_1 &= 1/4 \sum_{i=0}^{3} x_i y_{1\oplus i} \\ &= 1/4[x_0y_1 + x_1y_0 + x_2y_3 + x_3y_2] \\ &= 1/4[1{\cdot}7 + 5{\cdot}2 + 3{\cdot}1 + 8{\cdot}4] \\ &= 13.00 \end{aligned}$$

For $\tau = 2$ the convolution result is

$$\begin{aligned} z_2 &= 1/4 \sum_{i=0}^{3} x_i y_{2\oplus i} \\ &= 1/4[x_0y_2 + x_1y_3 + x_2y_0 + x_3y_1] \\ &= 1/4[1{\cdot}4 + 5{\cdot}1 + 3{\cdot}2 + 8{\cdot}7] \\ &= 17.75 \end{aligned}$$

Finally, for $\tau = 3$

$$\begin{aligned} z_3 &= 1/4 \sum_{i=0}^{3} x_i y_{3\oplus i} \\ &= 1/4[x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0] \\ &= 1/4[1{\cdot}1 + 5{\cdot}4 + 3{\cdot}7 + 8{\cdot}2] \\ &= 14.50 \end{aligned}$$

The first step, in applying the DWT to perform the dyadic convolution of x_i and y_i, is to perform the DWT on both sequences. Using the matrix vector form of the transform, the 4-point Walsh matrix is given by

$$W_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \tag{A.7}$$

The DWT then is given by

$$X_n = (1/4)W_4 x_i \tag{A.8}$$

For a general 4-point sequence, written as a column vector

$$x_i^T = [A \;\; B \;\; C \;\; D]$$

such that the 4-point DWT is given by

$$X_n = 1/4 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} \tag{A.9}$$

The resulting DWT is given by

$$X_n = 1/4 \begin{bmatrix} A + B + C + D \\ A + B - C - D \\ A - B - C + D \\ A - B + C - D \end{bmatrix}$$

Substituting in the x_i and y_i sequence values for A,B.C,D yields a DWT for x_i of

$$X_n = 1/4 \begin{bmatrix} 17 \\ -5 \\ 1 \\ -9 \end{bmatrix}$$

and for y_i of

$$Y_n = 1/4 \begin{bmatrix} 14 \\ 4 \\ -8 \\ -2 \end{bmatrix}$$

Performing the multiplication of Y_n and X_n yields

$$Z_n = \begin{bmatrix} 14.875 \\ -1.25 \\ -0.50 \\ 1.125 \end{bmatrix}$$

The inverse DWT of Z_n is evaluated by multiplying the 4×4 matrix W_4 and 4×1 vector Z_n

$$z_r = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 14.875 \\ -1.25 \\ -0.50 \\ 1.125 \end{bmatrix} \tag{A.10}$$

which yields the 4×1 vector

$$z_r = \begin{bmatrix} 14.25 \\ 13.00 \\ 17.75 \\ 14.50 \end{bmatrix} \tag{A.11}$$

This result is equivalent to the result calculated using the dyadic convolution sum.

Utilizing the FFT instead of the DWT in this example results in circular convolution of the two sequences. Linear convolution is performed if both sequences are zero end-padded with 4 zeros. The result of zero padding the x_i and y_i sequences and taking the inverse DWT of the product of their transforms, produces the result in Equation A.11 multiplied by 1/2. The two sequences in this case are

$$\begin{aligned} x_i &= \{1\ 5\ 3\ 8\ 0\ 0\ 0\ 0\} \\ y_i &= \{2\ 7\ 4\ 1\ 0\ 0\ 0\ 0\} \end{aligned}$$

The required 8-point DWT Walsh matrix is given by

$$W_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \end{bmatrix} \qquad \text{(A.12)}$$

The resulting DWT for y_i is

$$Y_n = \begin{bmatrix} 1.75 \\ 1.75 \\ 0.5 \\ 0.5 \\ -1 \\ -1 \\ -0.25 \\ -0.25 \end{bmatrix}$$

The resulting DWT for x_i is

$$X_n = \begin{bmatrix} 2.125 \\ 2.125 \\ -0.625 \\ -0.625 \\ 0.125 \\ 0.125 \\ -1.125 \\ -1.125 \end{bmatrix}$$

Multiplying X_n and Y_n and taking the inverse DWT produces

$$z_T = \begin{bmatrix} 7.125 \\ 6.5 \\ 8.875 \\ 7.25 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

which is the result in Equation A.11 scaled by 1/2 and end-padded with 4 zeros.

A.3 DWT of Time-shifted periodic signals

A.3.1 Sinusoid. The effects of shifting a sinusoid on the resulting DWT are presented in this section. The sinusoidal signal used has a period of 16 samples. DWT spectra of one and two periods of this signal, shifted and unshifted, are presented for one (Section A.3.1.1) and two periods (Section A.3.1.2) of the signal. Notationally an n-point shifted sinusoid will be referred to as the $n - shift$ sinusoid.

A.3.1.1 16-Point Transform. The 16-point DWT of the signal in Figure A.2 is represented for zero, single, two, three, and four sample shifts in Figures A.3, A.4, A.5, A.6, and A.7. Clearly, the shift variant nature of the DWT is demonstrated in that the 16-point DWT of this signal is changing as the signal is shifted.

A.3.1.2 32-Point Transform. The 32-point DWT of the signal in Figure A.2 is represented for zero, single, two, three, and four sample shifts in Figures A.8, A.9, A.10, A.11, and A.12. Notationally an n-point shifted sinusoid will be referred to as the $n - shift$ sinusoid.

Clearly, the 32- point DWT of this signal is changing as the signal is shifted. Also, the spectrums are clearly different from those in Figures A.3, A.4, A.5, A.6, and A.7, because the DWT does not assume periodicity of the input.

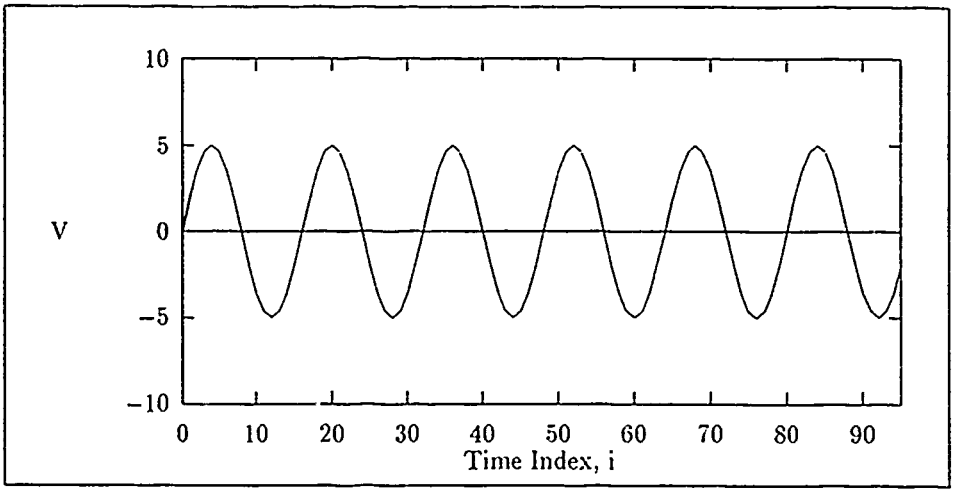

Figure A.2. This is a sinusoid with 16 sample period and amplitude of 5

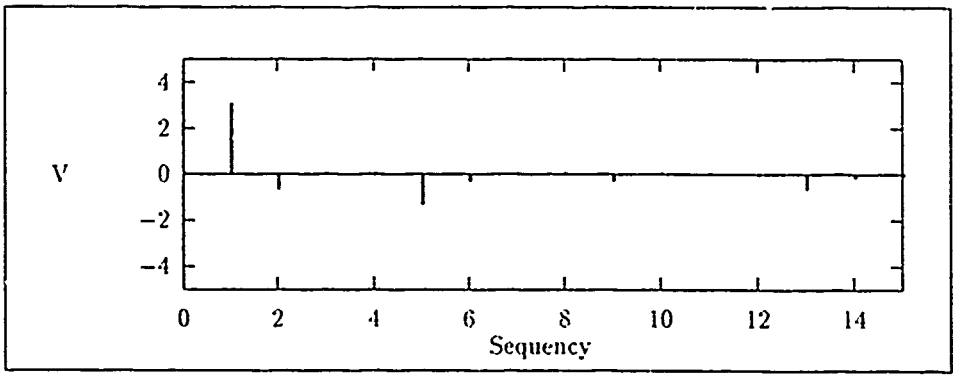

Figure A.3. This is the DWT of one period of the $0 - shift$ sinusoid.

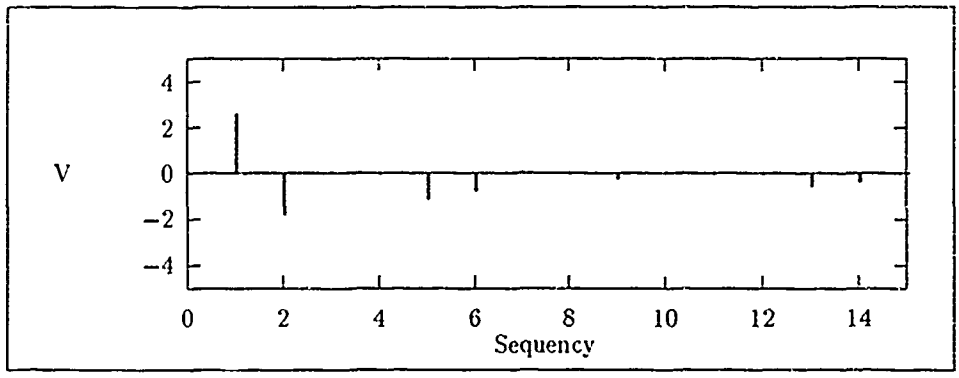

Figure A.4. This is the DWT of one period of the $1 - shift$ sinusoid

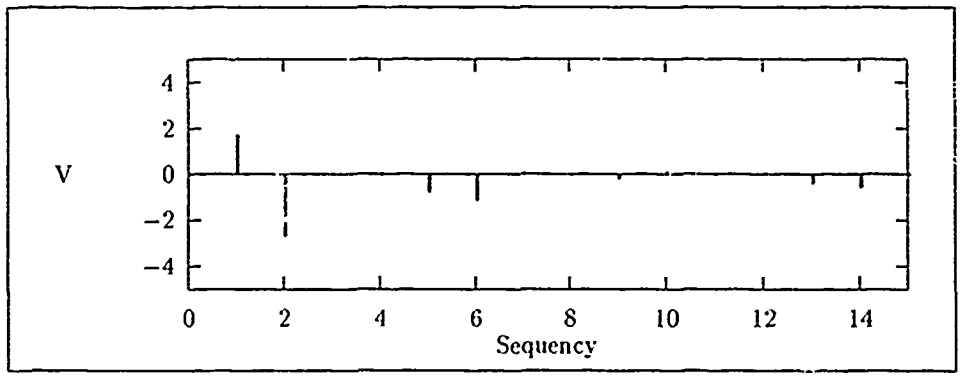

Figure A.5. This is the DWT of one period of the $2 - shift$ sinusoid.

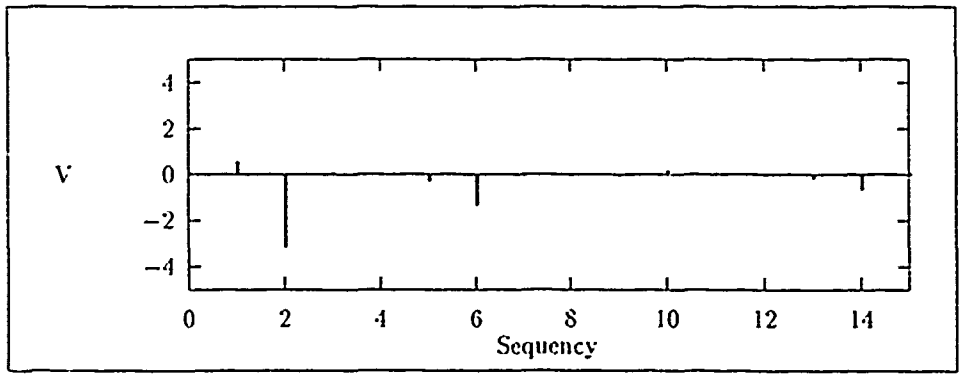

Figure A.6. This is the DWT of one period of the $3 - shift$ sinusoid.

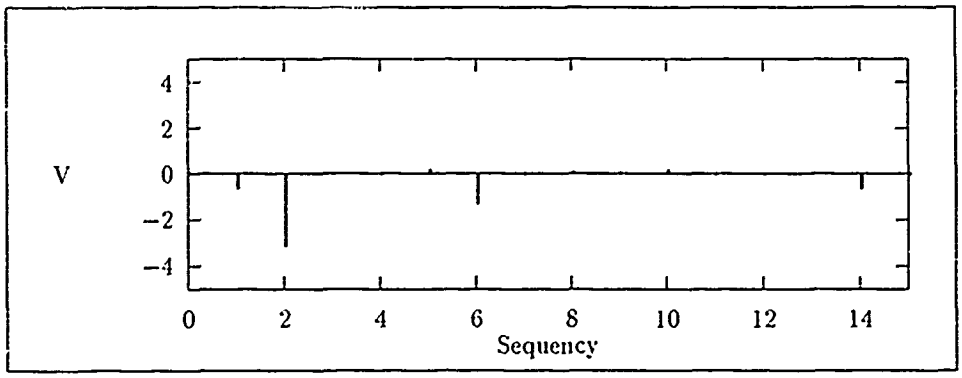

Figure A.7. This is the DWT of one period of the $4 - shift$ sinusoid.

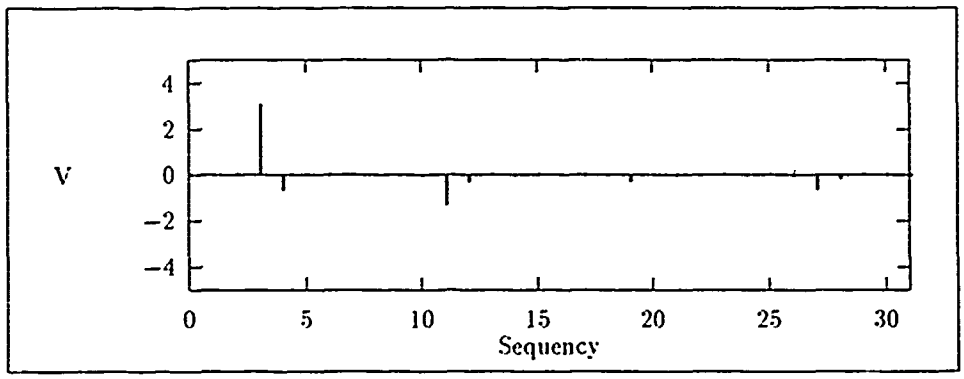

Figure A.8. This is the DWT of two periods of the $0 - shift$ sinusoid.

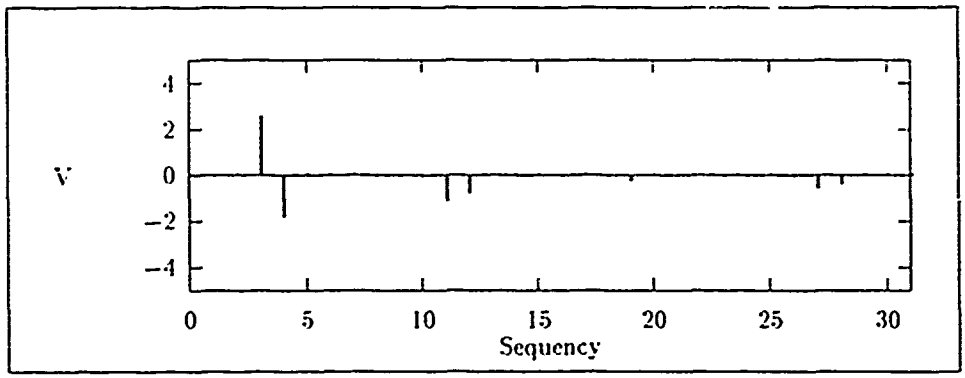

Figure A.9. This is the DWT of two periods of the $1 - shift$ sinusoid

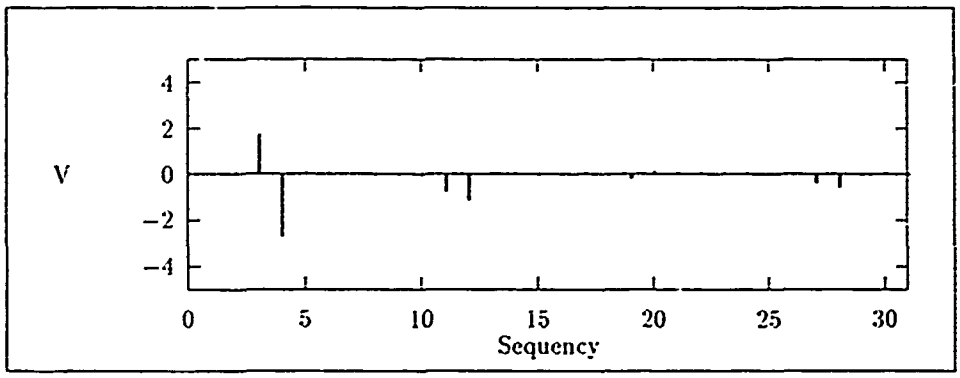

Figure A.10. This is the DWT of two periods of the $2 - shift$ sinusoid.

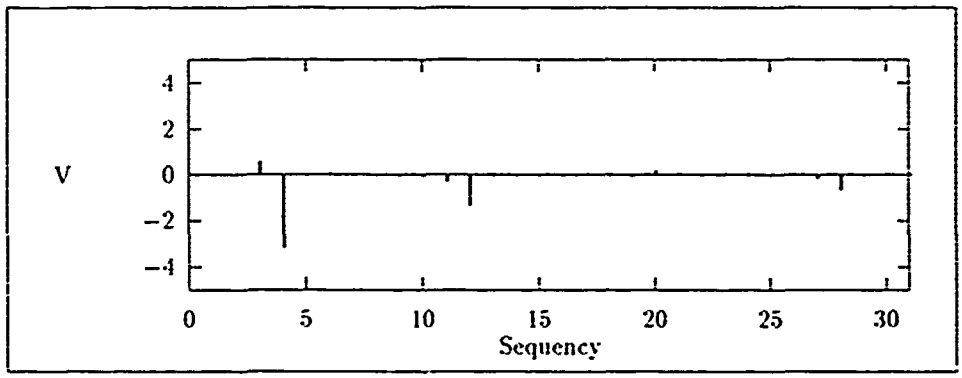

Figure A.11. This is the DWT of two periods of the $3 - shift$ sinusoid.

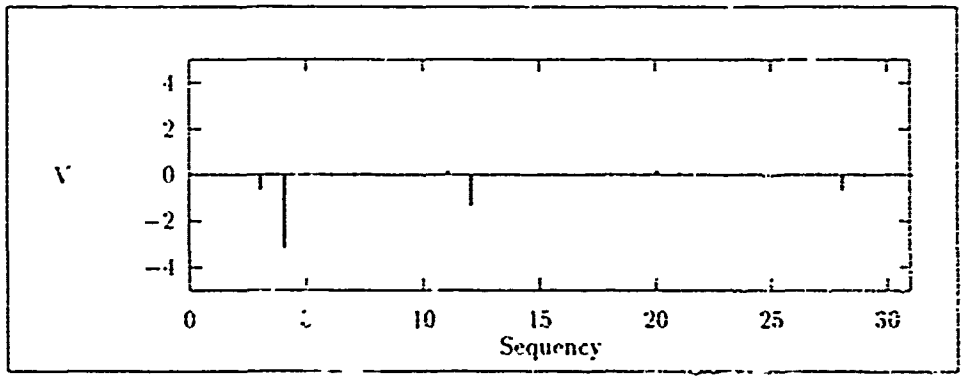

Figure A.12. This is the DWT of two periods of the $4 - shift$ sinusoid.

A.3.2 Rectangular. The effects of shifting a rectangular signal, on the resulting DWT are presented in this section. The rectangular signal used has a period of 16 samples. DWT spectra of one and two periods of this signal, shifted and unshifted, are presented for one (Section A.3.2.1) and two periods (Section A.3.2.2) of the signal. Notationally an n-point shifted rectangular signal will be referred to as the $n - shift$ rectangular signal.

A.3.2.1 16-Point Transform. The 16-point DWT of the signal in Figure A.13 is represented for zero, single, two, three, and four sample shifts in Figures A.14, A.15, A.16, A.17, and A.18. Clearly, the 16- point DWT of this signal is changing as the signal is shifted, which again demonstrates the shift variant nature of the DWT.

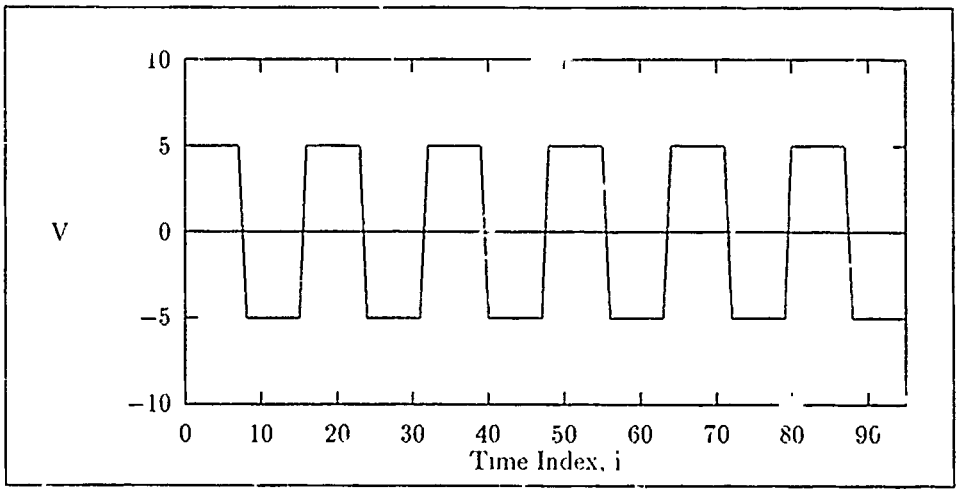

Figure A.13. This is a rectangular signal with 16 sample period and amplitude of 5

A.3.2.2 32-Point Transform. The 32-point DWT of the signal in Figure A.13 is represented for zero, single, two, three, and four sample shifts in Figures A.19, A.20, A.21, A.22, and A.23. Clearly, the 32- point DWT of this signal is changing as the signal is shifted.

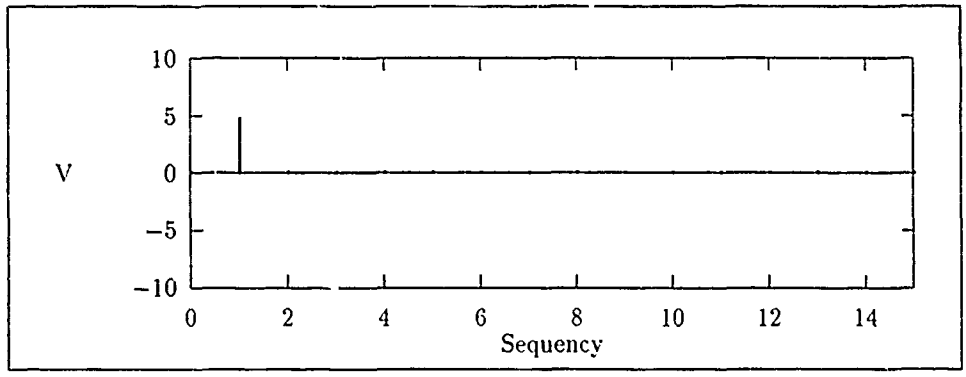

Figure A.14. This is the DWT of one period of the $0 - shift$ rectangular signal.

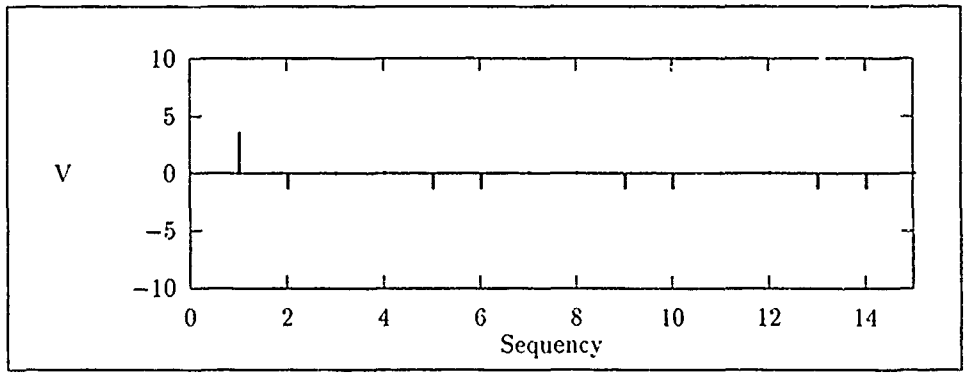

Figure A.15. This is the DWT of one period of the $1 - shift$ rectangular signal.

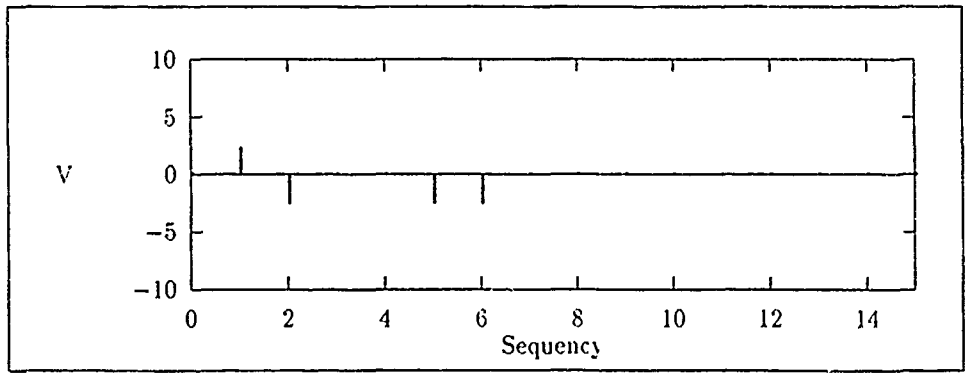

Figure A.16. This is the DWT of one period of the $2 - shift$ rectangular signal.

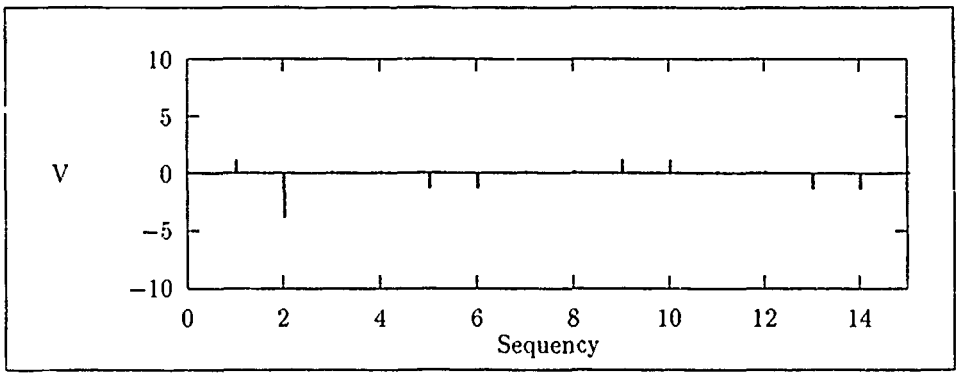

Figure A.17. This is the DWT of one period of the $3 - shift$ rectangular signal.

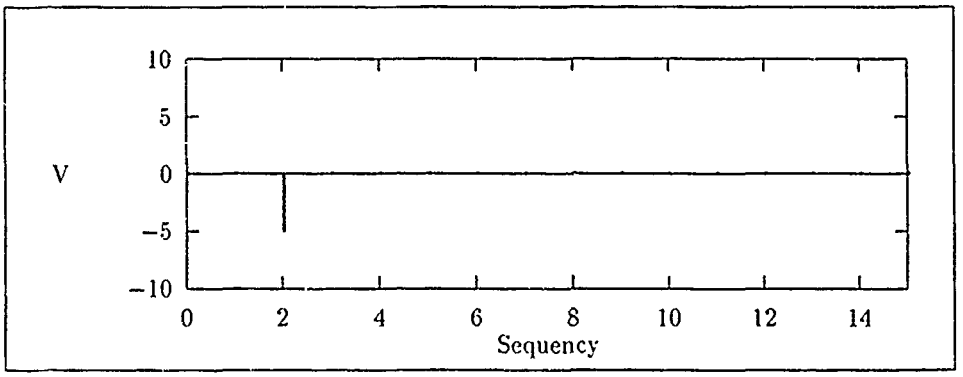

Figure A.18. This is the DWT of one period of the $4 - shift$ rectangular signal.

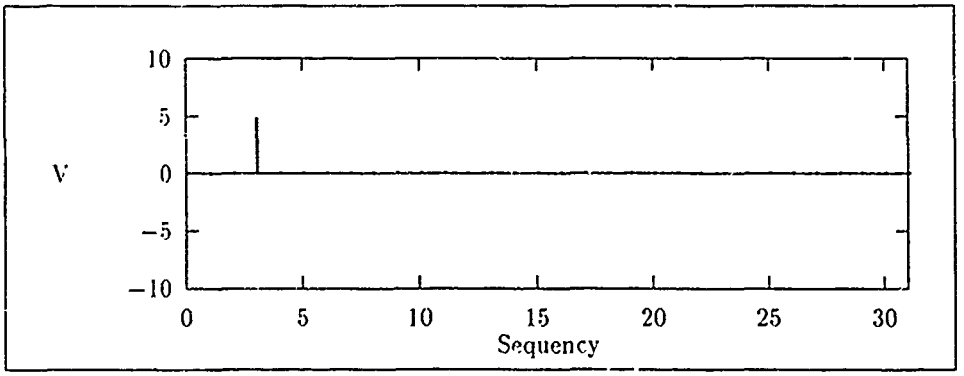

Figure A.19. This is the DWT of two periods of the $0 - shift$ rectangular signal.

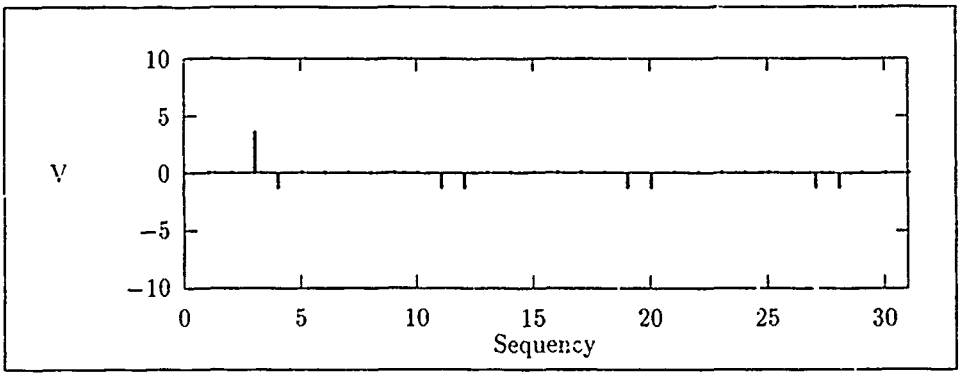

Figure A.20. This is the DWT of two periods of the $1 - shift$ rectangular signal.

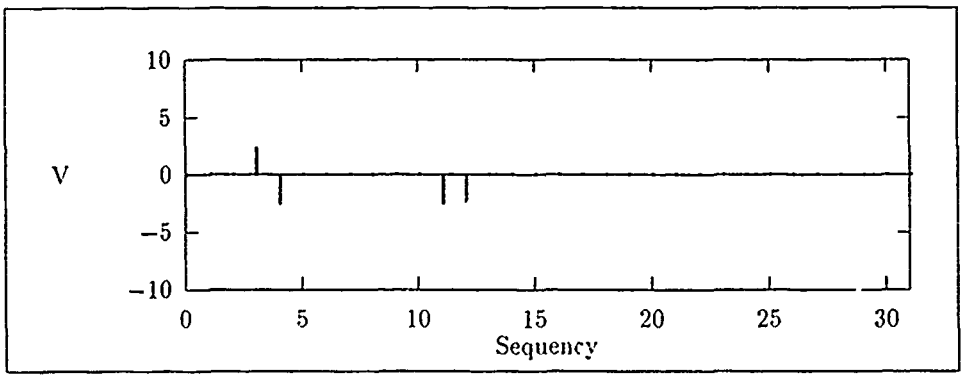

Figure A.21. This is the DWT of two periods of the $2 - shift$ rectangular signal.

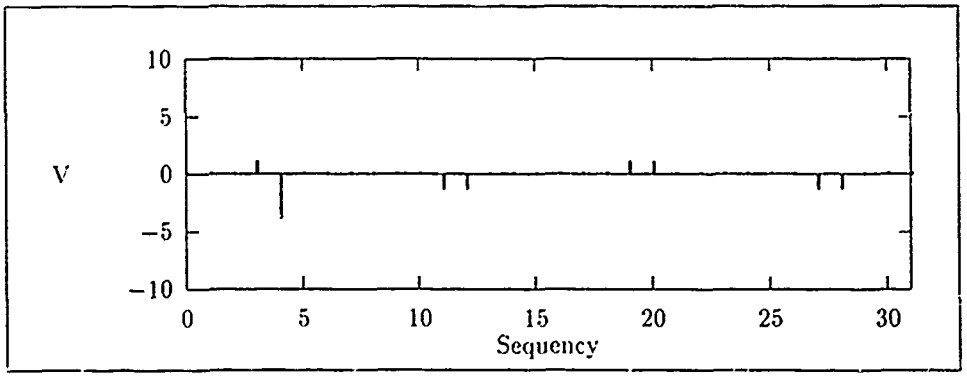

Figure A.22. This is the DWT of two period of the $3 - shift$ rectangular signal.

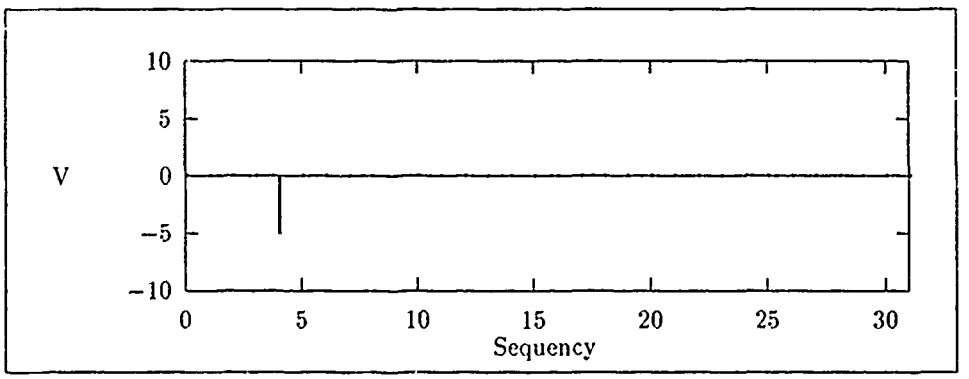

Figure A.23. This is the DWT of two periods of the $4 - shift$ rectangular signal.

Appendix B. *WDF1 Weight Update*

This Appendix derives the WDF1 Walsh-domain gradient vector which is used in Chapter II (see Section 3.1). An example is presented for the $N = 4$ block size. A WDF1 time-domain gradient example is also presented for the block $k = 2$ using the block size $N = 4$. This example is used to facilitate a comparison with the WDF2, FDF1, and FDF2 filters.

B.1 Walsh-Domain Gradient

The Walsh-domain weight vector is specified by

$$\mathrm{H}(k+1) = \mathrm{H}(k) + \mu \mathrm{X}(k)\mathrm{E}(k) \tag{B.1}$$

where the $\mathrm{X}(k)$, $\mathrm{E}(k)$, and $\mathrm{H}(k)$ vector components are real valued quantities since the Discrete Walsh Transform (DWT) produces real valued components. The product $\mathrm{X}(k)\mathrm{E}(k)$ in Equation B.1 defines the Walsh-Domain gradient vector $\nabla_{W1}(k)$ for WDF1. Using $\nabla_{W1}(k)$, Equation B.1 can be expressed as

$$\begin{bmatrix} H_0(k+1) \\ H_1(k+1) \\ \vdots \\ H_{N-1}(k+1) \end{bmatrix} = \begin{bmatrix} H_0(k) \\ H_1(k) \\ \vdots \\ H_{N-1}(k) \end{bmatrix} + \mu \begin{bmatrix} \nabla_{W1_0}(k) \\ \nabla_{W1_1}(k) \\ \vdots \\ \nabla_{W1_{N-1}}(k) \end{bmatrix} \tag{B.2}$$

The first task in deriving the Walsh-domain gradient vector $\nabla_{W1}(k)$ is to define some necessary time-domain notation. Using the notation x_i to represent the input sequence, the N input sequence values which define the kth input block can be represented by x_{kN+i}, where $i = 0, 1, \ldots, N-1$. Using this notation, the N-point kth block input vector is defined

$$\mathrm{x}(k) = [x_{kN} \quad \ldots \quad x_{kN+N-1}] \tag{B.3}$$

Applying the same notation to the desired time-domain sequence, the associated kth block desired vector is represented as

$$\mathrm{d}(k) = [d_{kN} \quad \ldots \quad d_{kN+N-1}] \tag{B.4}$$

The kth block input DWT matrix X(k) is given by

$$\mathrm{X}(k) = diag\{\mathcal{W}\{[x_{(kN)} \quad \ldots \quad x_{(kN+N-1)}]\}\} \tag{B.5}$$

where $\mathcal{W}$ is the forward DWT operator. The forward and inverse N-point DWT pair [1:50] is as follows:

$$X_n = 1/N \sum_{i=0}^{N-1} x_i WAL(n,i) \tag{B.6}$$

and

$$x_i = \sum_{n=0}^{N-1} X_n WAL(n,i) \tag{B.7}$$

The DWT components of the kth N-point input block, $X_n(k)$, comprise the diagonal components of the input matrix as indicated in the following equation

$$\mathrm{X}(k) = \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_N(k) \end{bmatrix} \tag{B.8}$$

The Walsh-domain output vector for the kth block is calculated as follows:

$$\mathrm{Y}(k) = \mathrm{X}(k)\mathrm{H}(k) \tag{B.9}$$

so that

$$\mathrm{Y}(k) = \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{N-1}(k) \end{bmatrix} \begin{bmatrix} H_0(k) \\ H_1(k) \\ \vdots \\ H_{N-1}(k) \end{bmatrix}$$

The resulting Walsh-Domain output vector is then

$$\mathrm{Y}(k) = \begin{bmatrix} X_0(k)H_0(k) \\ X_1(k)H_1(k) \\ \vdots \\ X_{N-1}(k)H_{N-1}(k) \end{bmatrix} \tag{B.10}$$

The Walsh-domain error vector for the kth block is defined

$$\mathrm{E}(k) = \mathrm{D}(k) - \mathrm{Y}(k) \tag{B.11}$$

where $\mathrm{D}(k)$ is the kth block desired DWT component vector. Therefore, $\mathrm{D}(k)$ is defined

$$\mathrm{D}(k) = \mathcal{W}\{[d_{(kN)} \quad \ldots \quad d_{(kN+N-1)}]\} \tag{B.12}$$

and

$$\mathrm{D}(k) = \begin{bmatrix} D_0(k) \\ D_1(k) \\ \vdots \\ D_{N-1}(k) \end{bmatrix}$$

Substituting Equations B.1 and B.10 into Equation B.11 results in

$$\mathrm{E}(k) = \begin{bmatrix} D_0(k) - X_0(k)H_0(k) \\ D_1(k) - X_1(k)H_1(k) \\ \vdots \\ D_{N-1}(k) - X_{N-1}(k)H_{N-1}(k) \end{bmatrix} \tag{B.13}$$

Performing the matrix vector multiplication $\mathrm{X}(k)\mathrm{E}(k)$ produces

$$\nabla_{W1}(k) = \begin{bmatrix} \nabla_{W1_0}(k) \\ \nabla_{W1_1}(k) \\ \vdots \\ \nabla_{W1_{N-1}}(k) \end{bmatrix} = \begin{bmatrix} X_0(k)[D_0(k) - X_0(k)H_0(k)] \\ X_1(k)[D_1(k) - X_1(k)H_1(k)] \\ \vdots \\ X_{N-1}(k)[D_{N-1}(k) - X_{N-1}(k)H_{N-1}(k)] \end{bmatrix} \tag{B.14}$$

This is the result used in Section 3.1.

B.2 Time-domain Gradient

The equivalent time-domain gradient expression for WDF1 is given by

$$\nabla_j(k) = 1/N \sum_{i=0}^{N-1} e_i(k)x_{(i\oplus j)}(k), \quad j = 0, 1, \ldots, N-1 \tag{B.15}$$

where j is the time-domain weight index, $\oplus$ indicates modulo-2 addition for the binary representations of i and j, and $\nabla_j(k)$ defines the kth block gradient term for each time-domain weight. The $x_i(k)$ and $e_i(k)$ terms represent the ith component of $\mathrm{x}(k)$ and $\mathrm{e}(k)$ respectively, during the kth block:

$$\begin{aligned} \mathrm{x}(k) &= [x_{kN} \ \ldots \ x_{kN+N-1}] \\ &= [x_0(k) \ \ldots \ x_{(N-1)}(k)] \end{aligned} \tag{B.16}$$

and

$$\begin{aligned} \mathrm{e}(k) &= [e_{kN} \ \ldots \ e_{kN+N-1}] \\ &= [e_0(k) \ \ldots \ e_{(N-1)}(k)] \end{aligned} \tag{B.17}$$

The first step in defining the $e_i(k)$ terms is to define $y_i(k)$. The $y_i(k)$ terms are derived from the inverse DWT (Equation B.7) of $\mathrm{Y}(k)$ (Equation B.10):

$$\mathrm{y}(k) = W^{-1}\{\mathrm{Y}(k)\} \tag{B.18}$$

where $\mathcal{W}^{-1}$ is the inverse DWT operator. Notationally, the N output sequence values for the kth block are represented by y_{kN+i}, where $i = 0, 1, \ldots, N-1$. Therefore, the kth block output vector y(k) is represented by

$$\mathrm{y}(k) = [y_{kN} \quad \ldots \quad y_{kN+N-1}]^T \tag{B.19}$$

The $e_i(k)$ terms represent the ith component of the time-domain error vector e(k) such that

$$\begin{aligned} \mathrm{e}(k) &= \mathcal{W}^{-1}\{\mathrm{E}(k)\} \\ &= [(d_{kN} - y_{kN}) \quad \ldots \quad (d_{kN+N-1} - y_{kN+N-1})]^T \\ &= [e_0(k) \quad \ldots \quad e_{N-1}(k)]^T \end{aligned} \tag{B.20}$$

Using $\nabla_j(k)$, the time domain weight update equation can be specified

$$h_j(k+1) = h_j(k) + \mu \nabla_j(k) \tag{B.21}$$

so that

$$\begin{bmatrix} h_0(k+1) \\ h_1(k+1) \\ \vdots \\ h_{N-1}(k+1) \end{bmatrix} = \begin{bmatrix} h_0(k) \\ h_1(k) \\ \vdots \\ h_{N-1}(k) \end{bmatrix} + \mu \begin{bmatrix} \nabla_0(k) \\ \nabla_1(k) \\ \vdots \\ \nabla_{N-1}(k) \end{bmatrix} \tag{B.22}$$

For a $N = 4$ block size, the expression for the $k = 2$ block, where in general $k = 0, 1, \ldots$, becomes

$$\begin{bmatrix} h_0(3) \\ h_1(3) \\ h_2(3) \\ h_3(3) \end{bmatrix} = \begin{bmatrix} h_0(2) \\ h_1(2) \\ h_2(2) \\ h_3(2) \end{bmatrix} + \mu \begin{bmatrix} \nabla_0(2) \\ \nabla_1(2) \\ \nabla_2(2) \\ \nabla_3(2) \end{bmatrix} \tag{B.23}$$

Each gradient term is calculated using

$$\nabla_j(2) = 1/4 \sum_{i=0}^{3} e_i(2) x_{(i \oplus j)}(2), \quad j = 0, 1, \ldots, 3 \tag{B.24}$$

where

$$\begin{aligned} \mathrm{x}(k) &= [x_8 \;\; x_9 \;\; x_{10} \;\; x_{11}] \\ \mathrm{e}(k) &= [e_8 \;\; e_9 \;\; e_{10} \;\; e_{11}] \end{aligned}$$

The initial sample index for both $\mathrm{x}(k)$ and $\mathrm{e}(k)$ begins at $i = 8$ since $kN = 2 \times 4$.

The $j = 0$ term is

$$\begin{aligned} \nabla_0(2) &= 1/4[e_0(2)x_0(2) + e_1(2)x_1(2) + e_2(2)x_2(2) + e_3(2)x_3(2)] \\ &= 1/4[e_8x_8 + e_9x_9 + e_{10}x_{10} + e_{11}x_{11}] \end{aligned}$$

The $j = 1$ term is

$$\begin{aligned} \nabla_1(2) &= 1/4[e_0(2)x_1(2) + e_1(2)x_0(2) + e_2(2)x_3(2) + e_3(2)x_2(2)] \\ &= 1/4[e_8x_9 + e_9x_8 + e_{10}x_{11} + e_{11}x_{10}] \end{aligned}$$

The $j = 2$ term is

$$\begin{aligned} \nabla_2(2) &= 1/4[e_0(2)x_2(2) + e_1(2)x_3(2) + e_2(2)x_0(2) + e_3(2)x_1(2)] \\ &= 1/4[e_8x_{10} + e_9x_{11} + e_{10}x_8 + e_{11}x_9] \end{aligned}$$

Finally, the $j = 3$ term is

$$\begin{aligned} \nabla_3(2) &= 1/4[e_0(2)x_3(2) + e_1(2)x_2(2) + e_2(2)x_1(2) + e_3(2)x_0(2)] \\ &= 1/4[e_8x_{11} + e_9x_{10} + e_{10}x_9 + e_{11}x_8] \end{aligned}$$

Each of the $\nabla_j(2)$ terms represents the dyadic convolution of the kth error block and input block.

Appendix C. *WDF2 Weight Update*

This Appendix derives the WDF2 Walsh-domain gradient vector . An example is presented for the $N = 2$ block size. A WDF2 time-domain gradient example is also presented for the block $k = 2$ using the block size $N = 4$.

C.1 Walsh-Domain Gradient

The Walsh-domain WDF2 weight vector is specified by

$$\mathrm{H}(k+1) = \mathrm{H}(k) + \mu \mathrm{X}(k)\mathrm{E}(k) \tag{C.1}$$

where the $\mathrm{X}(k)$, $\mathrm{E}(k)$, and $\mathrm{H}(k)$ vector components are real valued quantities and the product $\mathrm{X}(k)\mathrm{E}(k)$ defines the WDF2 gradient vector $\nabla_{W2}(k)$. The WDF2 filter requires the use of $2N$-point transforms due to the 50% overlap method employed in determining the input vector. Accordingly, the filter uses $2N$ Walsh-domain taps to filter the data. In vector form the weight update equation can be expressed as

$$\begin{bmatrix} H_0(k+1) \\ H_1(k+1) \\ \vdots \\ H_{2N-1}(k+1) \end{bmatrix} = \begin{bmatrix} H_0(k) \\ H_1(k) \\ \vdots \\ H_{2N-1}(k) \end{bmatrix} + \mu \begin{bmatrix} \nabla_{W2_0}(k) \\ \nabla_{W2_1}(k) \\ \vdots \\ \nabla_{W2_{2N-1}}(k) \end{bmatrix} \tag{C.2}$$

At this point we begin the $\nabla_{W2}(k)$ derivation by defining the kth block WDF2 time-domain input vector $\mathrm{x}(k)$. Using the notation x_i to represent the input sequence, the N input sequence values which define the kth input block can be represented by x_{kN+i}, where $i = 0, 1, \ldots, N-1$. Next, the N-point previous block and N-point current block are concatenated to produce the kth block input vector, defined as

$$\mathrm{x}(k) = [\underbrace{x_{(kN-N)} \quad \cdots \quad x_{(kN-1)}}_{(k-1)th\ block} \quad \underbrace{x_{(kN)} \quad \cdots \quad x_{(k.+N-1)}}_{kth\ block}] \tag{C.3}$$

The kth block input transform matrix $\mathrm{X}(k)$ is given by

$$\mathrm{X}(k) = diag\{\mathcal{W}[\underbrace{x_{(kN-N)} \quad \cdots \quad x_{(kN-1)}}_{(k-1)th\ block} \quad \underbrace{x_{(kN)} \quad \cdots \quad x_{(kN+N-1)}}_{kth\ block}]\} \tag{C.4}$$

where $\mathcal{W}$ is the forward DWT operator. The forward and inverse N-point DWT pair [1:50] is as follows:

$$X_n = 1/N \sum_{i=0}^{N-1} x_i WAL(n, i) \tag{C.5}$$

and

$$x_i = \sum_{n=0}^{N-1} X_n WAL(n, i) \tag{C.6}$$

Representing the DWT transform components of the $2N$-point kth block input vector as $X_n(k)$, $\mathrm{X}(k)$ can be expressed as

$$\mathrm{X}(k) = \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{2N-1}(k) \end{bmatrix} \tag{C.7}$$

The first step in deriving the Walsh-domain expression for $\mathrm{E}(k)$, is the evaluation of the kth block Walsh-domain output vector $\mathrm{Y}(k)$:

$$\mathrm{Y}(k) = \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{2N-1}(k) \end{bmatrix} \begin{bmatrix} H_0(k) \\ H_1(k) \\ \vdots \\ H_{2N-1}(k) \end{bmatrix} \tag{C.8}$$

Performing the multiplication of the $2N \times 2N$ matrix $X(k)$ and $2N \times 1$ vector $H(k)$ yields the $2N \times 1$ vector $Y(k)$ given by

$$Y(k) = \begin{bmatrix} X_0(k)H_0(k) \\ X_1(k)H_1(k) \\ \vdots \\ X_{2N-1}(k)H_{2N-1}(k) \end{bmatrix} \tag{C.9}$$

Taking the inverse DWT of $Y(k)$ and saving the last N values (See Section 3.2) produces the kth block output vector. Representing the N output sequence values which define the kth output block as y_{kN+i}, where $i = 0, 1, \ldots, N-1$, the kth block time-domain output block vector is defined

$$\begin{aligned} y(k) &= [y_{(kN)}, \ldots, y_{(kN+N-1)}]^T \\ &= \textit{last } N \textit{ terms of } W^{-1}\{X(k)H(k)\} \end{aligned} \tag{C.10}$$

The W^{-1} operator used in Equation C.10 is the inverse DWT operator. Using the inverse DWT sum (Equation C.6), the $y(k)$ vector can be expressed as

$$y(k) = \begin{bmatrix} y_{kN} \\ y_{kN+1} \\ \vdots \\ y_{kN+N-1} \end{bmatrix} = \begin{bmatrix} \sum_{n=0}^{2N-1} WAL(n, N)X_n(k)H_n(k) \\ \sum_{n=0}^{2N-1} WAL(n, N+1)X_n(k)H_n(k) \\ \vdots \\ \sum_{n=0}^{2N-1} WAL(n, 2N-1)X_n(k)H_n(k) \end{bmatrix} \tag{C.11}$$

The Walsh-domain error vector $E(k)$ for the kth block is defined by

$$E(k) = W\{[\underbrace{0 \quad \ldots \quad 0}_{N \text{ zeros}} \quad \underbrace{(d_{(kN)} - y_{(kN)}) \quad \ldots \quad (d_{(kN+N-1)} - y_{(kN+N-1)})}_{k\text{th error block}}]^T\} \tag{C.12}$$

where $d_{(kN+i)}$ represents the ith sample in the kth N-point desired sequence block.

Using the linear property of the DWT (See Section 2.1.2), Equation C.12 can be expressed as

$$\begin{aligned} \mathrm{E}(k) \;=\; & \mathcal{W}\{[\underbrace{0 \;\ldots\; 0}_{N\ zeros}\; d_{(kN)} \;\cdots\; d_{(kN+N-1)}]^T\} \\ & -\mathcal{W}\{[\underbrace{0 \;\ldots\; 0}_{N\ zeros}\; y_{(kN)} \;\cdots\; y_{(kN+N-1)}]^T\} \end{aligned} \tag{C.13}$$

Evaluation of

$$\mathcal{W}\{[\underbrace{0 \;\ldots\; 0}_{N\ zeros}\; y_{(kN)} \;\cdots\; y_{(kN+N-1)}]^T\} \tag{C.14}$$

using Equations C.5and C.11 produces the vector

$$\frac{1}{2N}\begin{bmatrix} \sum_{i=N}^{2N-1}\sum_{n=0}^{2N-1} WAL(0,i)WAL(n,i)X_n(k)H_n(k) \\ \sum_{i=N}^{2N-1}\sum_{n=0}^{2N-1} WAL(1,i)WAL(n,i)X_n(k)H_n(k) \\ \vdots \\ \sum_{i=N}^{2N-1}\sum_{n=0}^{2N-1} WAL(2N-1,i)WAL(n,i)X_n(k)H_n(k) \end{bmatrix} \tag{C.15}$$

Cancellation of terms reduces this expression to

$$1/2\begin{bmatrix} X_0(k)H_0(k) - X_1(k)H_1(k) \\ -X_0(k)H_0(k) + X_1(k)H_1(k) \\ X_2(k)H_2(k) - X_3(k)H_3(k) \\ -X_2(k)H_2(k) + X_3(k)H_3(k) \\ \vdots \\ X_{(2N-4)}(k)H_{(2N-4)}(k) - X_{(2N-3)}(k)H_{(2N-3)}(k) \\ -X_{(2N-4)}(k)H_{(2N-4)}(k) + X_{(2N-3)}(k)H_{(2N-3)}(k) \\ X_{(2N-2)}(k)H_{(2N-2)}(k) - X_{(2N-1)}(k)H_{(2N-1)}(k) \\ -X_{(2N-2)}(k)H_{(2N-2)}(k) + X_{(2N-1)}(k)H_{(2N-1)}(k) \end{bmatrix} \tag{C.16}$$

where $(1/2N)\cdot N = 1/2$. For N greater than 2

$$\mathcal{W}[\underbrace{0 \ \ldots \ 0}_{N \text{ zeros}} \ d_{(kN)} \ \ldots \ d_{(kN+N-1)}]^T \tag{C.17}$$

can be expressed as

$$1/2 \begin{bmatrix} D_{N_0}(k) \\ -D_{N_0}(k) \\ -D_{N_1}(k) \\ D_{N_1}(k) \\ \vdots \\ D_{N_{N-2}}(k) \\ -D_{N_{N-2}}(k) \\ -D_{N_{N-1}}(k) \\ D_{N_{N-1}}(k) \end{bmatrix} \tag{C.18}$$

where the $D_{N_n}(k)$ terms represent the nth component of the N-point DWT of the current N-point block of the desired signal. The 1/2 factor results because the N-point DWT terms were calculated using a $2N$-point DWT. The N-point DWT components of the desired signal occur due to the zero front-padding of the vector transformed. For $N = 2$ Term C.17 produces the vector

$$1/4 \begin{bmatrix} d_{(2k)} + d_{(2k+1)} \\ -d_{(2k)} - d_{(2k+1)} \\ -d_{(2k)} + d_{(2k+1)} \\ d_{(2k)} - d_{(2k+1)} \end{bmatrix} \tag{C.19}$$

Substituting the vector results C.18 and C.16 into Equation C.13, results in

$$\mathrm{E}(k) = 1/2 \begin{bmatrix} D_{N_0}(k) \\ -D_{N_0}(k) \\ -D_{N_1}(k) \\ D_{N_1}(k) \\ \vdots \\ D_{N_{N-2}}(k) \\ -D_{N_{N-2}}(k) \\ -D_{N_{N-1}}(k) \\ D_{N_{N-1}}(k) \end{bmatrix} - 1/2 \begin{bmatrix} X_0(k)H_0(k) - X_1(k)H_1(k) \\ -X_0(k)H_0(k) + X_1(k)H_1(k) \\ X_2(k)H_2(k) - X_3(k)H_3(k) \\ -X_2(k)H_2(k) + X_3(k)H_3(k) \\ \vdots \\ X_{(2N-4)}(k)H_{(2N-4)}(k) - X_{(2N-3)}(k)H_{(2N-3)}(k) \\ -X_{(2N-4)}(k)H_{(2N-4)}(k) + X_{(2N-3)}(k)H_{(2N-3)}(k) \\ X_{(2N-2)}(k)H_{(2N-2)}(k) - X_{(2N-1)}(k)H_{(2N-1)}(k) \\ -X_{(2N-2)}(k)H_{(2N-2)}(k) + X_{(2N-1)}(k)H_{(2N-1)}(k) \end{bmatrix} \tag{C.20}$$

which simplifies to

$$\mathrm{E}(k) = 1/2 \begin{bmatrix} D_{N_0}(k) - X_0(k)H_0(k) + X_1(k)H_1(k) \\ -D_{N_0}(k) + X_0(k)H_0(k) - X_1(k)H_1(k) \\ -D_{N_1}(k) - X_2(k)H_2(k) + X_3(k)H_3(k) \\ D_{N_1}(k) + X_2(k)H_2(k) - X_3(k)H_3(k) \\ \vdots \\ D_{N_{N-2}}(k) - X_{(2N-4)}(k)H_{(2N-4)}(k) + X_{(2N-3)}(k)H_{(2N-3)}(k) \\ -D_{N_{N-2}}(k) + X_{(2N-4)}(k)H_{(2N-4)}(k) - X_{(2N-3)}(k)H_{(2N-3)}(k) \\ -D_{N_{N-1}}(k) - X_{(2N-2)}(k)H_{(2N-2)}(k) + X_{(2N-1)}(k)H_{(2N-1)}(k) \\ D_{N_{N-1}}(k) + X_{(2N-2)}(k)H_{(2N-2)}(k) - X_{(2N-1)}(k)H_{(2N-1)}(k) \end{bmatrix} \tag{C.21}$$

Performing the multiplication of the $2N \times 2N$ matrix X(k) and $2N \times 1$ vector E(k) yields the $2N \times 1$ vector $\nabla_{W2}(k)$

$$\nabla_{W2}(k) = \begin{bmatrix} \nabla_{W2_0}(k) \\ \nabla_{W2_1}(k) \\ \nabla_{W2_2}(k) \\ \nabla_{W2_3}(k) \\ \vdots \\ \nabla_{W2_{2N-4}}(k) \\ \nabla_{W2_{2N-3}}(k) \\ \nabla_{W2_{2N-2}}(k) \\ \nabla_{W2_{2N-1}}(k) \end{bmatrix} \tag{C.22}$$

$$= 1/2 \begin{bmatrix} X_0(k)[D_{N_0}(k) - X_0(k)H_0(k) + X_1(k)H_1(k)] \\ X_1(k)[-D_{N_0}(k) + X_0(k)H_0(k) - X_1(k)H_1(k)] \\ X_2(k)[-D_{N_1}(k) - X_2(k)H_2(k) + X_3(k)H_3(k)] \\ X_3(k)[D_{N_1}(k) + X_2(k)H_2(k) - X_3(k)H_3(k)] \\ \vdots \\ X_{(2N-4)}(k)[D_{N_{N-2}}(k) - X_{(2N-4)}(k)H_{(2N-4)}(k) + X_{(2N-3)}(k)H_{(2N-3)}(k)] \\ X_{(2N-3)}(k)[-D_{N_{N-2}}(k) + X_{(2N-4)}(k)H_{(2N-4)}(k) - X_{(2N-3)}(k)H_{(2N-3)}(k)] \\ X_{(2N-2)}(k)[-D_{N_{N-1}}(k) - X_{(2N-2)}(k)H_{(2N-2)}(k) + X_{(2N-1)}(k)H_{(2N-1)}(k)] \\ X_{(2N-1)}(k)[D_{N_{N-1}}(k) + X_{(2N-2)}(k)H_{(2N-2)}(k) - X_{(2N-1)}(k)H_{(2N-1)}(k)] \end{bmatrix} \tag{C.23}$$

This is the result presented in Section 3.2. For $N = 2$ the Walsh-domain gradient is

$$\begin{bmatrix} \nabla_{W2_0}(k) \\ \nabla_{W2_1}(k) \\ \nabla_{W2_2}(k) \\ \nabla_{W2_3}(k) \end{bmatrix} = 1/4 \begin{bmatrix} X_0(k)[d_{(2k)} + d_{(2k+1)} - 2X_0(k)H_0(k) + 2X_1(k)H_1(k)] \\ X_1(k)[-d_{(2k)} - d_{(2k+1)} + 2X_0(k)H_0(k) - 2X_1(k)H_1(k)] \\ X_2(k)[-d_{(2k)} + d_{(2k+1)} - 2X_2(k)H_2(k) + 2X_3(k)H_3(k)] \\ X_3(k)[d_{(2k)} - d_{(2k+1)} + 2X_2(k)H_2(k) - 2X_3(k)H_3(k)] \end{bmatrix} \tag{C.24}$$

C.2 *Time-domain Gradient*

The equivalent time-domain gradient expression for WDF2 is given by

$$\nabla_j(k) = 1/2N \sum_{i=0}^{2N-1} e_i(k) x_{(i \oplus j)}(k), \quad j = 0, 1, \ldots, 2N-1 \tag{C.25}$$

where j is the time-domain weight index, $\oplus$ indicates modulo-2 addition for the binary representations of i and j, and $\nabla_j(k)$ defines the kth block gradient term for each time-domain weight. The $x_i(k)$ and $e_i(k)$ terms represent the ith component of x(k) and e(k) respectively, during the kth block:

$$\begin{aligned} \mathrm{x}(k) &= [\underbrace{x_{(kN-N)} \ \cdots \ x_{(kN-1)}}_{(k-1)th\ block} \ \underbrace{x_{(kN)} \ \cdots \ x_{(kN+N-1)}}_{kth\ block}] \\ &= [x_0(k) \ \cdots \ x_{(2N-1)}(k)] \end{aligned} \tag{C.26}$$

and

$$\begin{aligned} \mathrm{e}(k) &= [\underbrace{0 \ \cdots \ 0}_{N\ zeros} \ \underbrace{(d_{(kN)} - y_{(kN)}) \ \cdots \ (d_{(kN+N-1)} - y_{(kN+N-1)})}_{kth\ error\ block}]^T \\ &= [e_0(k) \ \cdots \ e_{(2N-1)}(k)]^T \end{aligned} \tag{C.27}$$

Using $\nabla_j(k)$, the time domain weight update equation can be specified

$$h_j(k+1) = h_j(k) + \mu \nabla_j(k) \tag{C.28}$$

so that

$$\begin{bmatrix} h_0(k+1) \\ h_1(k+1) \\ \vdots \\ h_{2N-1}(k+1) \end{bmatrix} = \begin{bmatrix} h_0(k) \\ h_1(k) \\ \vdots \\ h_{2N-1}(k) \end{bmatrix} + \mu \begin{bmatrix} \nabla_0(k) \\ \nabla_1(k) \\ \vdots \\ \nabla_{2N-1}(k) \end{bmatrix} \tag{C.29}$$

For a $N = 4$ block size, the expression for the $k = 2$ block becomes

$$\begin{bmatrix} h_0(3) \\ h_1(3) \\ \vdots \\ h_7(3) \end{bmatrix} = \begin{bmatrix} h_0(2) \\ h_1(2) \\ \vdots \\ h_7(2) \end{bmatrix} + \mu \begin{bmatrix} \nabla_0(2) \\ \nabla_1(2) \\ \vdots \\ \nabla_7(2) \end{bmatrix} \tag{C.30}$$

Each gradient term is calculated using

$$\nabla_j(2) = 1/8 \sum_{i=0}^{7} e_i(2) x_{(i \oplus j)}(2), \quad j = 0, 1, \ldots, 7 \tag{C.31}$$

where

$$\begin{aligned} \mathrm{x}(k) &= [x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9 \ x_{10} \ x_{11}] \\ \mathrm{e}(k) &= [0 \ 0 \ 0 \ 0 \ e_8 \ e_9 \ e_{10} \ e_{11}] \end{aligned}$$

The first sequence value in $\mathrm{x}(k)$ is x_4 because $k = 2$ and $N = 4$, so that $kN - N = 4$. Similarly, the first nonzero sequence value in $\mathrm{e}(k)$ is e_8 because $kN = 8$. Excluding the zero valued products in each sum, the time-domain gradient terms are as follows:

The $j = 0$ term is

$$\begin{aligned} \nabla_0(2) &= 1/8[e_4(2)x_4(2) + e_5(2)x_5(2) + e_6(2)x_6(2) + e_7(2)x_7(2)] \\ &= 1/8[e_8x_8 + e_9x_9 + e_{10}x_{10} + e_{11}x_{11}] \end{aligned}$$

The $j = 1$ term is

$$\begin{aligned} \nabla_1(2) &= 1/8[e_4(2)x_5(2) + e_5(2)x_4(2) + e_6(2)x_7(2) + e_7(2)x_6(2)] \\ &= 1/8[e_8x_9 + e_9x_8 + e_{10}x_{11} + e_{11}x_{10}] \end{aligned}$$

The $j = 2$ term is

$$\begin{aligned} \nabla_2(2) &= 1/8[e_4(2)x_6(2) + e_5(2)x_7(2) + e_6(2)x_4(2) + e_7(2)x_5(2)] \\ &= 1/8[e_8x_{10} + e_9x_{11} + e_{10}x_8 + e_{11}x_9] \end{aligned}$$

The $j = 3$ term is

$$\begin{aligned}\nabla_3(2) &= 1/8[e_4(2)x_7(2) + e_5(2)x_6(2) + e_6(2)x_5(2) + e_7(2)x_4(2)] \\ &= 1/8[e_8x_{11} + e_9x_{10} + e_{10}x_9 + e_{11}x_8]\end{aligned}$$

The $j = 4$ term is

$$\begin{aligned}\nabla_4(2) &= 1/8[e_4(2)x_0(2) + e_5(2)x_1(2) + e_6(2)x_2(2) + e_7(2)x_3(2)] \\ &= 1/8[e_8x_4 + e_9x_5 + e_{10}x_6 + e_{11}x_7]\end{aligned}$$

The $j = 5$ term is

$$\begin{aligned}\nabla_5(2) &= 1/8[e_4(2)x_1(2) + e_5(2)x_0(2) + e_6(2)x_3(2) + e_7(2)x_2(2)] \\ &= 1/8[e_8x_5 + e_9x_4 + e_{10}x_7 + e_{11}x_6]\end{aligned}$$

The $j = 6$ term is

$$\begin{aligned}\nabla_6(2) &= 1/8[e_4(2)x_2(2) + e_5(2)x_3(2) + e_6(2)x_0(2) + e_7(2)x_1(2)] \\ &= 1/8[e_8x_6 + e_9x_7 + e_{10}x_4 + e_{11}x_5]\end{aligned}$$

Finally, the $j = 7$ term is

$$\begin{aligned}\nabla_7(2) &= 1/8[e_4(2)x_3(2) + e_5(2)x_2(2) + e_6(2)x_1(2) + e_7(2)x_0(2)] \\ &= 1/8[e_8x_7 + e_9x_6 + e_{10}x_5 + e_{11}x_4]\end{aligned}$$

A comparison of the gradient terms just calculated with those calculated for WDF1 in Appendix B, clearly reveals a difference. There are 8 WDF2 terms and only 4 WDF1 gradient terms. This is because the number of time-domain taps equals the transform size: WDF2 uses $2N$-point transforms and WDF1 uses N-point transforms. The first N WDF2 gradient values are equal to the WDF1 value multiplied by 1/2. The 1/2 factor results from the change in transform size.

Appendix D. *FDF1 Weight Update*

This Appendix derives the FDF1 frequency-domain gradient vector. An example is presented for the $N = 4$ block size. A FDF1 time-domain gradient example is also presented for the block $k = 2$ using the block size $N = 4$. This example is used to facilitate a comparison with WDF1, WDF2, and FDF2.

D.1 Frequency-Domain Gradient

The frequency-domain weight vector is specified by

$$\mathrm{H}(k+1) = \mathrm{H}(k) + \mu \mathrm{X}^{*}(k)\mathrm{E}(k) \tag{D.1}$$

where the $\mathrm{X}(k)$, $\mathrm{E}(k)$, and $\mathrm{H}(k)$ vector components are in general complex valued quantities. The $*$ notation indicates complex conjugate. The product $\mathrm{X}^{*}(k)\mathrm{E}(k)$ in Equation B.1 defines the frequency-domain gradient vector $\nabla_{F1}(k)$ for FDF1. Using $\nabla_{F1}(k)$, Equation D.1 can be expressed as

$$\begin{bmatrix} H_0(k+1) \\ H_1(k+1) \\ \vdots \\ H_{N-1}(k+1) \end{bmatrix} = \begin{bmatrix} H_0(k) \\ H_1(k) \\ \vdots \\ H_{N-1}(k) \end{bmatrix} + \mu \begin{bmatrix} \nabla_{F1_0}(k) \\ \nabla_{F1_1}(k) \\ \vdots \\ \nabla_{F1_{N-1}}(k) \end{bmatrix} \tag{D.2}$$

Given that the $\mathrm{X}(k)$, $\mathrm{E}(k)$, and $\mathrm{H}(k)$ vector components are in general complex, the weight update equation for the jth tap can be represented as two separate update operations:

$$H_{real_j}(k+1) = H_{real_j}(k) + \mu \nabla_{F1real_j}(k) \tag{D.3}$$

$$H_{imag_j}(k+1) = H_{imag_j}(k) + \mu \nabla_{F1imag_j}(k) \tag{D.4}$$

where $H_{real_j}(k)$ and $H_{imag_j}(k)$ represent the real and imaginary parts of $H_j(k)$. The terms $\nabla_{F1real_j}(k)$ and $\nabla_{F1imag_j}(k)$ represent the real and imaginary parts of $\nabla_{F1_j}(k)$. Therefore, during the kth block each frequency-domain tap is decomposed into a real component and an imaginary component. The respective kth block frequency-domain gradient term $\nabla_{F1_j}(k)$

is similarly decomposed and the real and imaginary kth block jth gradient terms are used to update the respective part of the $H_j(k)$ tap. The expansion of $\nabla_{F1}(k)$ in terms of the frequency-domain vectors X(k), H(k), and D(k) follows the same development presented in Appendix B for WDF1. All time-domain vector notation is the same. The frequency-domain vector notation follows the Walsh-domain vector notation with the exception of replacing the $\mathcal{W}$ and $\mathcal{W}^{-1}$ operators with the corresponding DFT operators $\mathcal{F}$ and $\mathcal{F}^{-1}$. Therefore, $\nabla_{F1}(k)$ has the same general form as the $\nabla_{W1}(k)$ result (Equation D.5), such that

$$\nabla_{F1}(k) = \begin{bmatrix} \nabla_{F1_0}(k) \\ \nabla_{F1_1}(k) \\ \vdots \\ \nabla_{F1_{N-1}}(k) \end{bmatrix} = \begin{bmatrix} X_0^*(k)[D_0(k) - X_0(k)H_0(k)] \\ X_1^*(k)[D_1(k) - X_1(k)H_1(k)] \\ \vdots \\ X_{N-1}^*(k)[D_{N-1}(k) - X_{N-1}(k)H_{N-1}(k)] \end{bmatrix} \tag{D.5}$$

The forward and inverse DFT pair used in the development is given by [5:150]

$$X_n = \sum_{i=0}^{N-1} x_i W_N^{ni} \tag{D.6}$$

and

$$x_i = 1/N \sum_{n=0}^{N-1} X_n W_N^{-ni} \tag{D.7}$$

where

$$W_N = e^{-j2\pi/N} \tag{D.8}$$

D.2 Time-domain Gradient

The equivalent time-domain gradient expression for FDF1 is given by

$$\nabla(k) = \sum_{i=0}^{N-1} e_i(k)x_i(k), \quad j = 0, 1, \ldots, N-1 \tag{D.9}$$

The $e_i(k)$ terms represent the ith component of the time-domain error vector $e(k)$ such that

$$\begin{aligned} e(k) &= \mathcal{F}^{-1}\{E(k)\} \\ &= [(d_{kN} - y_{kN}) \quad \cdots \quad (d_{kN+N-1} - y_{kN+N-1})]^T \end{aligned}$$

$$= [e_0(k) \ \ldots \ e_{N-1}(k)]^T \tag{D.10}$$

A brief discussion of the time-domain output vector is necessary to define $x_i(k)$. The time-domain kth block output vector is calculated by performing the circular convolution of the kth block time-domain weight vector $h(k)$ and the kth block input vector $x(k)$. Accordingly, the output vector is defined

$$y(k) = \chi(k)h(k) \tag{D.11}$$

where

$$\chi(k) = \begin{bmatrix} x_0(k) & x_{(N-1)}(k) & \ldots & x_1(k) \\ x_1(k) & x_0(k) & \ldots & x_2(k) \\ \vdots & \vdots & & \vdots \\ x_{N-1}(k) & x_{N-2}(k) & \ldots & x_0(k) \end{bmatrix} \tag{D.12}$$

and $x_i(k)$ specifies the ith value of $x(k)$ (See Equation 2.35) while $x_i^T(k)$ is the ith row of $\chi(k)$.

Using $\nabla_j(k)$ to denote the jth component of $\nabla(k)$, the time domain weight update equation can be specified

$$h_j(k+1) = h_j(k) + \mu\nabla_j(k) \tag{D.13}$$

so that

$$\begin{bmatrix} h_0(k+1) \\ h_1(k+1) \\ \vdots \\ h_{N-1}(k+1) \end{bmatrix} = \begin{bmatrix} h_0(k) \\ h_1(k) \\ \vdots \\ h_{N-1}(k) \end{bmatrix} + \mu \begin{bmatrix} \nabla_0(k) \\ \nabla_1(k) \\ \vdots \\ \nabla_{N-1}(k) \end{bmatrix} \tag{D.14}$$

For a $N = 4$ block size, the expression for the $k = 2$ block becomes

$$\begin{bmatrix} h_0(3) \\ h_1(3) \\ h_2(3) \\ h_3(3) \end{bmatrix} = \begin{bmatrix} h_0(2) \\ h_1(2) \\ h_2(2) \\ h_3(2) \end{bmatrix} + \mu \begin{bmatrix} \nabla_0(2) \\ \nabla_1(2) \\ \nabla_2(2) \\ \nabla_3(2) \end{bmatrix} \tag{D.15}$$

Each gradient term is calculated using

$$\nabla_j(2) = \sum_{i=0}^{3} e_i(2)\mathrm{x}_i(2), \quad j = 0, 1, \ldots, 3 \tag{D.16}$$

where

$$\chi(k) = \begin{bmatrix} x_8 & x_{11} & x_{10} & x_9 \\ x_9 & x_8 & x_{11} & x_{10} \\ x_{10} & x_9 & x_8 & x_{11} \\ x_{11} & x_{10} & x_9 & x_8 \end{bmatrix}$$

$$\mathrm{e}(k) = [e_8 \;\; e_9 \;\; e_{10} \;\; e_{11}]$$

The $x_0(k)$ term is x_8 because $k = 2$ and $N = 4$ so that $kN = 8$. The $j = 0$ term is

$$\begin{aligned} \nabla_0(2) &= [e_0(2)x_0(2) + e_1(2)x_1(2) + e_2(2)x_2(2) + e_3(2)x_3(2)] \\ &= [e_8x_8 + e_9x_9 + e_{10}x_{10} + e_{11}x_{11}] \end{aligned}$$

The $j = 1$ term is

$$\begin{aligned} \nabla_1(2) &= [e_0(2)x_3(2) + e_1(2)x_0(2) + e_2(2)x_1(2) + e_3(2)x_2(2)] \\ &= [e_8x_{11} + e_9x_8 + e_{10}x_9 + e_{11}x_{10}] \end{aligned}$$

The $j = 2$ term is

$$\begin{aligned} \nabla_2(2) &= [e_0(2)x_2(2) + e_1(2)x_3(2) + e_2(2)x_0(2) + e_3(2)x_1(2)] \\ &= [e_8x_{10} + e_9x_{11} + e_{10}x_8 + e_{11}x_9] \end{aligned}$$

Finally, the $j = 3$ term is

$$\begin{aligned} \nabla_3(2) &= [e_0(2)x_1(2) + e_1(2)x_2(2) + e_2(2)x_3(2) + e_3(2)x_0(2)] \\ &= [e_8x_9 + e_9x_{10} + e_{10}x_{11} + e_{11}x_8] \end{aligned}$$

A comparison of the terms calculated above with similar calculations for WDF1 in Appendix B reveals that the first gradient term for both are equal. Subsequent terms have individual products that match products in corresponding WDF1 gradient terms.

Appendix E. *FDF2 Weight Update*

This Appendix derives the FDF2 frequency-domain gradient vector. An example is presented for the $N = 2$ block size. A FDF2 time-domain gradient example is also presented for the block $k = 2$ using the block size $N = 4$ and is used to facilitate a comparison with FDF1.

E.1 Frequency-Domain Gradient

The frequency-domain FDF2 weight vector is specified by

$$\begin{aligned} \mathrm{H}(k+1) &= \mathrm{H}(k) + \mu\mathcal{F}\begin{bmatrix} \nabla(k) \\ 0 \\ \vdots \\ 0 \end{bmatrix} \\ &= \mathrm{H}(k) + \mu\nabla_{F2}(k) \end{aligned} \tag{E.1}$$

where $\mathcal{F}$ is the forward FFT operator and the vertical vector is zero padded with N zeros. $\nabla(k)$ is defined

$$\nabla(k) = \textit{first } N \textit{ terms of } \mathcal{F}^{-1}\{\mathrm{X}^*(k)\mathrm{E}(k)\} \tag{E.2}$$

Referencing Equations E.1 and E.2, $\mathrm{X}(k)$, $\mathrm{E}(k)$, and $\mathrm{H}(k)$ vector components are, in general, complex valued quantities. The vector $\mathrm{X}^*(k)$ is the complex conjugate of $\mathrm{X}(k)$. The resultant vertical vector in Equation E.1 defines the FDF2 gradient vector $\nabla_{F2}(k)$. The FDF2 filter requires the use of $2N$-point transforms due to the 50% overlap method employed in determining the input vector. Accordingly, the filter uses $2N$ frequency-domain taps to filter the data. In vector form, the weight update equation can be expressed as

$$\begin{bmatrix} H_0(k+1) \\ H_1(k+1) \\ \vdots \\ H_{2N-1}(k+1) \end{bmatrix} = \begin{bmatrix} H_0(k) \\ H_1(k) \\ \vdots \\ H_{2N-1}(k) \end{bmatrix} + \mu \begin{bmatrix} \nabla_{F2_0}(k) \\ \nabla_{F2_1}(k) \\ \vdots \\ \nabla_{F2_{2N-1}}(k) \end{bmatrix} \tag{E.3}$$

Given that the X(k), E(k), and H(k) vector components are generally complex, the weight update equation for the jth tap can be represented as two separate update operations:

$$H_{real_j}(k+1) = H_{real_j}(k) + \mu \nabla_{F1real_j}(k) \quad \text{(E.4)}$$

$$H_{imag_j}(k+1) = H_{imag_j}(k) + \mu \nabla_{F1imag_j}(k) \quad \text{(E.5)}$$

where $H_{real_j}(k)$ and $H_{imag_j}(k)$ represent the real and imaginary parts of $H_j(k)$. The terms $\nabla_{F1real_j}(k)$ and $\nabla_{F1imag_j}(k)$ represent the real and imaginary parts of $\nabla_{F1_j}(k)$. Therefore, during the kth block each frequency-domain tap is decomposed into a real component and an imaginary component. The respective kth block frequency-domain gradient term $\nabla_{F2_j}(k)$ is similarly decomposed and the real and imaginary kth block jth gradient terms are used to update the respective part of the $H_j(k)$ tap.

At this point we begin the $\nabla_{F2}(k)$ derivation by defining the kth block FDF2 time-domain input vector x(k). Using the notation x_i to represent the input sequence, the N input sequence values which define the kth input block can be represented by x_{kN+i}, where $i = 0, 1, \ldots, N-1$. Next, the N-point previous block and N-point current block are concatenated to produce the kth block input vector, defined as

$$\mathrm{x}(k) = [\underbrace{x_{(kN-N)} \quad \cdots \quad x_{(kN-1)}}_{(k-1)th\ block} \quad \underbrace{x_{(kN)} \quad \cdots \quad x_{(kN+N-1)}}_{kth\ block}] \quad \text{(E.6)}$$

The kth block input transform matrix X(k) is given by

$$\mathrm{X}(k) = diag\{\mathcal{F}[\underbrace{x_{(kN-N)} \quad \cdots \quad x_{(kN-1)}}_{(k-1)th\ block} \quad \underbrace{x_{(kN)} \quad \cdots \quad x_{(kN+N-1)}}_{kth\ block}]\} \quad \text{(E.7)}$$

where $\mathcal{F}$ is the forward DFT operator. The forward and inverse $2N$-point DFT pair is defined [5:150]

$$X_n = \sum_{i=0}^{2N-1} x_i W_{2N}^{ni} \quad \text{(E.8)}$$

and

$$x_i = 1/2N \sum_{n=0}^{2N-1} X_n W_{2N}^{-ni} \quad \text{(E.9)}$$

where

$$W_{2N} = e^{-j2\pi/2N} \quad \text{(E.10)}$$

Representing the DFT transform components of the $2N$-point kth block input vector as $X_n(k)$, $\mathrm{X}(k)$ can be expressed as

$$\mathrm{X}(k) = \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{2N-1}(k) \end{bmatrix} \tag{E.11}$$

The first step in deriving the frequency-domain expression for $\mathrm{E}(k)$, is the evaluation of the kth block frequency-domain output vector $\mathrm{Y}(k)$:

$$\mathrm{Y}(k) = \begin{bmatrix} X_0(k) & & 0 \\ & X_1(k) & \\ & \ddots & \\ 0 & & X_{2N-1}(k) \end{bmatrix} \begin{bmatrix} H_0(k) \\ H_1(k) \\ \vdots \\ H_{2N-1}(k) \end{bmatrix} \tag{E.12}$$

Performing the multiplication of the $2N \times 2N$ matrix $\mathrm{X}(k)$ and $2N \times 1$ vector $\mathrm{H}(k)$ yields the $2N \times 1$ vector $\mathrm{Y}(k)$ given by

$$\mathrm{Y}(k) = \begin{bmatrix} X_0(k)H_0(k) \\ X_1(k)H_1(k) \\ \vdots \\ X_{2N-1}(k)H_{2N-1}(k) \end{bmatrix} \tag{E.13}$$

Taking the inverse DFT of $\mathrm{Y}(k)$ and saving the last N values (See Section 2.2.2.2) produces the kth block output vector. Representing the N output sequence values which define the kth output block as y_{kN+i}, where $i = 0, 1, \ldots N-1$, the kth block time-domain output block vector is defined

$$\begin{aligned} y(k) &= [y_{(kN)}, \ldots, y_{(kN+N-1)}]^T \\ &= \textit{last } N \textit{ terms of } \mathcal{F}^{-1}\{\mathrm{X}(k)\mathrm{H}(k)\} \end{aligned} \tag{E.14}$$

The $\mathcal{F}^{-1}$ operator used in Equation E.14 is the inverse DFT operator. Using the inverse DFT sum (Equation E.9), the y(k) vector can be expressed as

$$\mathrm{y}(k) = \frac{1}{2N}\begin{bmatrix} y_{kN} \\ y_{kN+1} \\ \vdots \\ y_{kN+N-1} \end{bmatrix} = \begin{bmatrix} \sum_{n=0}^{2N-1} W_{2N}^{-Nn} X_n(k) H_n(k) \\ \sum_{n=0}^{2N-1} W_{2N}^{-(N+1)n} X_n(k) H_n(k) \\ \vdots \\ \sum_{n=0}^{2N-1} W_{2N}^{-(2N-1)n} X_n(k) H_n(k) \end{bmatrix} \tag{E.15}$$

The frequency-domain error vector E(k) for the kth block is defined by

$$\mathrm{E}(k) = \mathcal{F}\{[\underbrace{0 \ \ldots \ 0}_{N \text{ zeros}} \ \underbrace{(d_{(kN)} - y_{(kN)}) \ \ldots \ (d_{(kN+N-1)} - y_{(kN+N-1)})}_{k\text{th error block}}]^T\} \tag{E.16}$$

where the $d_{(kN+i)}$ terms represent the kth N-point desired sequence block. Using the linear property of the DFT, Equation E.16 can be expressed as

$$\begin{aligned} \mathrm{E}(k) &= \mathcal{F}\{[\underbrace{0 \ \ldots \ 0}_{N \text{ zeros}} \ d_{(kN)} \ \ldots \ d_{(kN+N-1)}]^T\} \\ &\quad -\mathcal{F}\{[\underbrace{0 \ \ldots \ 0}_{N \text{ zeros}} \ y_{(kN)} \ \ldots \ y_{(kN+N-1)}]^T\} \\ &= \mathrm{E}_1(k) - \mathrm{E}_2(k) \end{aligned} \tag{E.17}$$

Evaluation of

$$\mathrm{E}_2(k) = \mathcal{F}\{[\underbrace{0 \ \ldots \ 0}_{N \text{ zeros}} \ y_{(kN)} \ \ldots \ y_{(kN+N-1)}]^T\} \tag{E.18}$$

using Equations E.8and E.15 produces the vector

$$\mathrm{E}_2(k) = \frac{1}{2N}\begin{bmatrix} \sum_{i=N}^{2N-1} \sum_{n=0}^{2N-1} W_{2N}^{-in} X_n(k) H_n(k) \\ \sum_{i=N}^{2N-1} \sum_{n=0}^{2N-1} W_{2N}^{i} W_{2N}^{-in} X_n(k) H_n(k) \\ \vdots \\ \sum_{i=N}^{2N-1} \sum_{n=0}^{2N-1} W_{2N}^{(2N-1)i} W_{2N}^{-in} X_n(k) H_n(k) \end{bmatrix} \tag{E.19}$$

The rth component of Equation E.19 can be expressed as

$$E_{2_r}(k) = 1/2N \sum_{i=N}^{2N-1} \sum_{n=0}^{2N-1} W_{2N}^{ri} W_{2N}^{-in} X_n(k) H_n(k), \qquad r = 0, 1, \ldots, 2N-1 \tag{E.20}$$

Evaluation of

$$\mathrm{E}_1(k) = \mathcal{F}[\underbrace{0 \quad \ldots \quad 0}_{N \text{ zeros}} \quad d_{(kN)} \quad \ldots \quad d_{(kN+N-1)}]^T \tag{E.21}$$

produces

$$E_{1_r}(k) = 1/2N \sum_{i=N}^{2N-1} W_{2N}^{ri} d_{(kN+i-N)}, \qquad r = 0, 1, \ldots, 2N-1 \tag{E.22}$$

such that

$$\mathrm{X}^*(k)\mathrm{E}(k) = \mathrm{X}^*(k)E_1(k) - \mathrm{X}^*(k)E_2(k) \tag{E.23}$$

At this time. the inverse DFT is performed on Equation E.23. Taking the inverse DFT of the product involving $E_1(k)$ produces

$$\nabla_{1_p}(k) = 1/2N \sum_{r=0}^{2N-1} E_{1_r}(k) X_r^*(k) W_{2N}^{-rp} \tag{E.24}$$

Substituting the result from Equation E.22 for $E_{1_r}(k)$ gives

$$\nabla_{1_p}(k) = 1/2N \sum_{r=0}^{2N-1} \sum_{i=N}^{2N-1} W_{2N}^{ri} W_{2N}^{-rp} X_r^*(k) d_{(kN+i-N)} \tag{E.25}$$

where $p = 0, 1, \ldots, 2N-1$. Taking the inverse DFT of the product involving $E_2(k)$ produces

$$\nabla_{2_p}(k) = 1/2N \sum_{r=0}^{2N-1} E_{2_r}(k) X_r^*(k) W_{2N}^{-rp} \tag{E.26}$$

and substitution of the result from Equation E.20 for $E_{2_r}(k)$ yields

$$\nabla_{2_p}(k) = (1/2N)^2 \sum_{r=0}^{2N-1} \sum_{i=N}^{2N-1} \sum_{n=0}^{2N-1} W_{2N}^{ri} W_{2N}^{-rp} W_{2N}^{-ni} X_r^*(k) X_n(k) H_n(k) \tag{E.27}$$

where $p = 0, 1, \ldots, 2N-1$. The next step in deriving $\nabla_{F2}(k)$ is to replace the last N values of $\nabla_2(k)$ and $\nabla_1(k)$ with zeros and forward transform the resulting vectors to produce two

frequency-domain vectors: identified as $\nabla_{F2_2}(k)$ and $\nabla_{F2_1}(k)$ respectively. $\nabla_{F2}(k)$ is defined by the difference of the resulting frequency-domain vectors:

$$\nabla_{F2}(k) = \nabla_{F2_1}(k) - \nabla_{F2_2}(k) \tag{E.28}$$

which can be expressed as

$$\begin{aligned}
\nabla_{F2_q}(k) &= 1/2N \sum_{p=0}^{N-1} \sum_{r=0}^{2N-1} \sum_{i=N}^{2N-1} W_{2N}^{ri} W_{2N}^{-rp} W_{2N}^{qp} X_r^*(k) d_{(kN+i-N)} \\
&\quad -(1/2N)^2 \sum_{p=0}^{N-1} \sum_{r=0}^{2N-1} \sum_{i=N}^{2N-1} \sum_{n=0}^{2N-1} W_{2N}^{ri} W_{2N}^{-rp} W_{2N}^{-ni} W_{2N}^{qp} X_r^*(k) X_n(k) H_n(k) \\
q &= 0, 1, \ldots, 2N-1
\end{aligned} \tag{E.29}$$

This is the solution presented in Chapter II.

For $N = 2$ the frequency-domain gradient real and imaginary (j) terms are as follows:

$$\begin{aligned}
\nabla_{F2_0}(k) &= (d_{(2k)}/4)[2X_0^*(k) - X_1^*(k)(1+j) - X_3^*(k)(1-j)] \\
&+(d_{(2k+1)}/4)[2X_0^*(k) - X_1^*(k)(1-j) - X_3^*(k)(1+j)] \\
&-|X_0(k)|^2 H_0(k) - (1/2)|X_1(k)|^2 H_1(k)(1+j) \\
&-(1/2)|X_3(k)|^2 H_3(k)(1-j) + (1/2)X_0(k)X_1^*(k)H_0(k) \\
&+(1/2)X_0(k)X_3^*(k)H_0(k) + (1/2)X_0^*(k)X_1(k)H_1(k)(1+j) \\
&+(j/2)X_1^*(k)X_2(k)H_2(k) - (j/2)X_3^*(k)X_2(k)H_2(k) \\
&+(1/2)X_0^*(k)X_3(k)H_3(k)(1-j)
\end{aligned}$$

$$\begin{aligned}
\nabla_{F2_1}(k) &= (d_{(2k)}/4)[X_0^*(k)(1-j) - 2X_1^*(k) + X_2^*(k)(1+j)] \\
&+(d_{(2k+1)}/4)[X_0^*(k)(1-j) - X_2^*(k)(1-j) + 2jX_1^*(k)] \\
&-|X_1(k)|^2 H_1(k) - (1/2)|X_0(k)|^2 H_0(k)(1-j) \\
&-(1/2)|X_2(k)|^2 H_2(k)(1-j) + (1/2)X_0(k)X_1^*(k)H_0(k)(1-j) \\
&+(1/2)X_0^*(k)X_1(k)H_1(k) + (1/2)X_2^*(k)X_1(k)H_1(k) \\
&+(1/2)X_1^*(k)X_2(k)H_2(k)(1+j) \\
&-(j/2)X_0^*(k)X_3(k)H_3(k) + (j/2)X_2^*(k)X_3(k)H_3(k)
\end{aligned}$$

$$\begin{aligned}
\nabla_{F2_2}(k) &= (d_{(2k)}/4)[-X_1^*(k)(1-j) + 2X_2^*(k) - X_3^*(k)(1+j)] \\
&+(d_{(2k+1)}/4)[-2X_2^*(k) + X_1^*(k)(1+j) + X_3^*(k)(1-j)] \\
&-|X_2(k)|^2 H_2(k) - (1/2)|X_1(k)|^2 H_1(k)(1-j) \\
&-(1/2)|X_3(k)|^2 H_3(k)(1+j) + (1/2)X_2(k)X_1^*(k)H_2(k) \\
&+(1/2)X_3^*(k)X_2(k)H_2(k) + (1/2)X_2^*(k)X_1(k)H_1(k)(1-j) \\
&+(1/2)X_2^*(k)X_3(k)H_0(k)(1+j) \\
&-(j/2)X_1^*(k)X_0(k)H_0(k) + (j/2)X_3^*(k)X_0(k)H_0(k)
\end{aligned}$$

$$
\begin{aligned}
\nabla_{F2_3}(k) \;=\; & (d_{(2k)}/4)[X_0^*(k)(1+j) + 2X_2^*(k)(1-j) - 2X_3^*(k)] \\
& +(d_{(2k+1)}/4)[X_0^*(k)(1+j) - X_2^*(k)(1-j) - 2jX_3^*(k)] \\
& -|X_3(k)|^2 H_3(k) - [illegible]X_0(k)|^2 H_0(k)(1+j) \\
& -(1/2)|X_2(k)|^2 H_{[illegible]}(k) + (1/2)X_0(k)X_3^*(k)H_0(k)(1+j) \\
& +(1/2)X_3^*(k)X_2(k)H_{[illegible]}(k)(1-j) + (1/2)X_0^*(k)X_3(k)H_3(k) \\
& +(1/2)X_2^*(k)X_3(k)H_{[illegible]}(k) \\
& +(j/2)X_0^*(k)X_1(k)H_{[illegible]}(k) - (j/2)X_2^*(k)X_1(k)H_1(k)
\end{aligned}
$$

E.2 Time-domain Gradient

The equivalent time-domain gradient expression for FDF2 is given by

$$\nabla_j(k) = \sum_{i=N}^{2N-1} e_i(k)x_{(i-j)}(k), \quad j = 0, 1, \ldots, N-1 \tag{E.30}$$

where j is the time-domain weight index and $\nabla_j(k)$ defines the kth block gradient term for each time-domain weight. The $x_i(k)$ and $e_i(k)$ terms represent the ith component of x(k) and e(k) respectively, in the kth block:

$$
\begin{aligned}
\mathrm{x}(k) \;&=\; [\underbrace{x_{(kN-N)} \;\ldots\; x_{(kN-1)}}_{(k-1)th\ block}\ \underbrace{x_{(kN)} \;\ldots\; x_{(kN+N-1)}}_{kth\ block}] \\
&=\; [x_0(k) \;\ldots\; x_{(2N-1)}(k)]
\end{aligned} \tag{E.31}
$$

and

$$
\begin{aligned}
\mathrm{e}(k) \;&=\; [\underbrace{0 \;\ldots\; 0}_{N\ zeros}\ \underbrace{(d_{(kN)} - y_{(kN)}) \;\ldots\; (d_{(kN+N-1)} - y_{(kN+N-1)})}_{kth\ error\ block}]^T \\
&=\; [e_0(k) \;\ldots\; e_{(2N-1)}(k)]^T
\end{aligned} \tag{E.32}
$$

Using $\nabla_j(k)$, the time-domain weight update equation can be specified

$$h_j(k+1) = h_j(k) + \mu\nabla_j(k) \tag{E.33}$$

so that

$$\begin{bmatrix} h_0(k+1) \\ h_1(k+1) \\ \vdots \\ h_{N-1}(k+1) \end{bmatrix} = \begin{bmatrix} h_0(k) \\ h_1(k) \\ \vdots \\ h_{N-1}(k) \end{bmatrix} + \mu \begin{bmatrix} \nabla_0(k) \\ \nabla_1(k) \\ \vdots \\ \nabla_{N-1}(k) \end{bmatrix} \tag{E.34}$$

For a $N = 4$ block size, the expression for the $k = 2$ block becomes

$$\begin{bmatrix} h_0(3) \\ h_1(3) \\ \vdots \\ h_3(3) \end{bmatrix} = \begin{bmatrix} h_0(2) \\ h_1(2) \\ \vdots \\ h_3(2) \end{bmatrix} + \mu \begin{bmatrix} \nabla_0(2) \\ \nabla_1(2) \\ \vdots \\ \nabla_3(2) \end{bmatrix} \tag{E.35}$$

Each gradient term is calculated using

$$\nabla_j(2) = \sum_{i=4}^{7} e_i(2) x_{(i-j)}(2), \quad j = 0, 1, \ldots, 3 \tag{E.36}$$

where

$$\begin{aligned} \mathrm{x}(k) &= [x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9 \ x_{10} \ x_{11}] \\ \mathrm{e}(k) &= [0 \ 0 \ 0 \ 0 \ e_8 \ e_9 \ e_{10} \ e_{11}] \end{aligned}$$

since $N = 4$, $k = 2$, and therefore the $\mathrm{x}(k)$ index $(kN - N + i)$ equals $(4 + i)$ and the $\mathrm{e}(k)$ index for the last N values, $(kN + i)$, equals $(8 + i)$.

Excluding the zero valued products in each sum, the time-domain gradient terms are as follows:

the $j = 0$ term is

$$\begin{aligned} \nabla_0(2) &= [e_4(2)x_4(2) + e_5(2)x_5(2) + e_6(2)x_6(2) + e_7(2)x_7(2)] \\ &= [e_8x_8 + e_9x_9 + e_{10}x_{10} + e_{11}x_{11}] \end{aligned}$$

The $j = 1$ term is

$$\begin{aligned}\nabla_1(2) &= [e_4(2)x_3(2) + e_5(2)x_4(2) + e_6(2)x_5(2) + e_7(2)x_6(2)] \\ &= [e_8x_7 + e_9x_8 + e_{10}x_9 + e_{11}x_{10}]\end{aligned}$$

The $j = 2$ term is

$$\begin{aligned}\nabla_2(2) &= [e_4(2)x_2(2) + e_5(2)x_3(2) + e_6(2)x_4(2) + e_7(2)x_5(2)] \\ &= [e_8x_6 + e_9x_7 + e_{10}x_8 + e_{11}x_9]\end{aligned}$$

Finally, the $j = 3$ term is

$$\begin{aligned}\nabla_3(2) &= [e_4(2)x_1(2) + e_5(2)x_2(2) + e_6(2)x_3(2) + e_7(2)x_4(2)] \\ &= [e_8x_5 + e_9x_6 + e_{10}x_7 + e_{11}x_8]\end{aligned}$$

In comparison to the FDF1 time-domain gradient terms (Appendix C), the FDF2 time-domain gradient terms above are the cross-correlation of x(k) and e(k). Since the FDF1 filter performs circular convolution, the gradient terms are calculated using the product of e(k) and the corresponding column of the kth block circulant matrix.

Appendix F. *Program Listings*

F.1 WDF1 Filter Listing

This is the Turbo Pascal 6.0 listing of the WDF1 program.

```
Program WDF1;
{$N+}
Uses Printer,CRT,DOS;
CONST
  N=8;          (*** N is block size*)
  P=3;         (* N=2**P *)
  datasize=1000;
  Misadjust=0.1;
TYPE

  RealArrayN2=ARRAY[0..(N-1)] of real;
  InputArray=ARRAY[0..datasize-1] of real;
  OutputArray=ARRAY[0..datasize-1] of real;
  FArray=ARRAY[0..P,1..N,0..N-1] of Real;
VAR
  hr,m,s,s100:Word;
  StartClock,StopClock:Real;
  data:RealArrayN2;
  F:FArray;
  gain_mu,Pbin,X,H,W,ERR,V2,input_block,grad,D,Yw:RealArrayN2;
  desired,input:InputArray;
  error,y:OutputArray;
  isign,nn,Block_num,k:integer;
  mu:real;
  infile,desiredfile,errorfile,outfile,weight0:text;
  weight1,weight2,weight3,weight4:text;
 (*************************************************************)
 (*                    Procedure Init_var                     *)
 (*                                                           *)
 (*    This procedure initializes all variables.              *)
 (*                                                           *)
 (*     Called By:                                            *)
 (*         Main Program                                      *)
 (*     Routines Called: None                                 *)
 (*************************************************************)
```

```
Procedure Init_var;
  Var
    l,j,i:integer;

    Begin

      Block_num:=0;
      For j:=0 to datasize-1 Do
        Begin
          error[j]:=0;
          desired[j]:=0;
          input[j]:=0;
          y[j]:=0;
        End;
      For j:=0 to (N-1) Do
         Begin
           X[j]:=0;
           W[j]:=0;
           H[j]:=0;
           ERR[j]:=0;
           V2[j]:=0;
           input_block[j]:=0;
           grad[j]:=0;
           gain_mu[j]:=0;
           Pbin[j]:=0;
           D[j]:=0;
           Yw[j]:=0;
        End;
     For j:=0 to P Do
       For i:=1 to N Do
         For l:=0 to N-1 Do
           F[j,i,l]:=0;


    End;

(*************************************************************)
(*                Procedure Open input files                 *)
(*                                                           *)
(*    Called By:                                             *)
(*         Main Program                                      *)
(*    Routines Called: None                                  *)
(*************************************************************)
Procedure Open_input_files;
   Begin
```

```
      Assign(infile, 'B:tstinpt2.Dat');
      Assign(desiredfile, 'B:tstdes2.Dat');
      Reset(infile);
      Reset(desiredfile);

   End;

(*****************************************************************)
(*                    Procedure Open output files                *)
(*                                                               *)
(*      Called By:                                               *)
(*         Main Program                                          *)
(*      Routines Called: None                                    *)
(*****************************************************************)
Procedure Open_output_files;
   Begin
      Assign(errorfile, 'B:Error.Dat');
      Rewrite(errorfile);
      Assign(outfile, 'B:Out.Dat');
      Rewrite(outfile);

   End;

(********* ***************************************************)
(*                  Procedure Close input files              *)
(*                                                           *)
(*     Called By:                                            *)
(*         Main Program                                      *)
(*     Routines Called: None                                 *)
(*************************************************************)
Procedure Close_input_files;
   Begin
      Close(infile);
      Close(desiredfile);

   End;

(*****************************************************************)
(*                Procedure Close output files                   *)
(*                                                               *)
(*      Called By:                                               *)
(*          Main Program                                         *)
(*      Routines Called: None                                    *)
(*****************************************************************)
Procedure Close_output_files;
```

```
  Begin
     Close(errorfile);
     Close(outfile);
  End;

 (*****************************************************************)
 Procedure ClockOn;
 Begin
   GetTime(hr,m,s,s100);
   StartClock:=(hr*3600)+(m*60)+s+(s100/100);
 End;
(*****************************************************************)
Procedure ClockOff;
 Begin
   GetTime(hr,m,s,s100);
   StopClock:=(hr*3600)+(m*60)+s+(s100/100);
   WriteLn('Elapsed time = ',(StopClock-StartClock):0:2);
 End;

(*****************************************************************)
(*                  Procedure Calc_numblocks                     *)
(*                                                               *)
(*     This procedure calculates the number of blocks to         *)
(*     be processed.                                             *)
(*                                                               *)
(*        Called By:                                             *)
(*           Main Program                                        *)
(*       Routines Called: None                                   *)
(*****************************************************************)

Procedure Calc_numblocks;
  Begin

   Block_num:=datasize div N;

  End;

 (*****************************************************************)
 (*                  Procedure Waltran                            *)
 (*                                                               *)
 (*   This procedure performs the forward and inverse N-point     *)
 (*   Fast Walsh Transform (FWT).  This routine calculates        *)
 (*   the forward and inverse transforms using the same loop:     *)
 (*   the forward transform requires the loop result to be        *)
 (*   multiplied by a factor of 1/(N).                            *)
```

```
(*   The algorithm is recursive and requires P stages, where     *)
(*   N=2**P.  This routine requires no multiplications           *)
(*   and NLog(N) summations, where the logarithm is base 2.      *)
(*                                                               *)
(*    Called By:                                                 *)
(*      Procedure Past_Current_Block                             *)
(*      Procedure Calculate_y                                    *)
(*      Procedure  Calculate_error                               *)
(*      Procedure  Calculate_Gradient                            *)
(*      Procedure Update_Weights                                 *)
(*    Routines Called:                                           *)
(*        None                                                   *)
(*    Variables:                                                 *)
(*           P: N=2**P                                           *)
(*      isign : indicates inverse or forward transform           *)
(*      data[j] : input and output                               *)
(*      F[P,1,j] :Fast Walsh Matrix result                       *)
(*             j=0,1,..2N-1                                      *)
(*****************************************************************)
(*****************************************************************)
Procedure Waltran;

  Var
     j,l,i,jmax,lmax,exponent:integer;
     jlog,jinv:real;

     Begin
      For l:=1 to N Do
        Begin
          F[0,1,0]:=data[l-1];
        (*  WriteLn(Lst, '  F(',0,1,0,')= ',F[0,1,0]); *)
        End;
      For i:=0 to P-1 Do
       Begin
        jlog:=(i+1)*Ln(2);
        jinv:=Exp(jlog);
        jmax:=Round(jinv);
   (*     WriteLn(Lst, 'jmax= ', jmax); *)
        lmax:=N Div jmax;
    (*    WriteLn(Lst, ' kmax= ', kmax); *)
        For l:=1 to lmax Do
          Begin
           For j:=0 to jmax-1 Do
             Begin
              exponent:=(j+1) Div 2;
```

```
            If Odd(exponent) then
              F[i+1,l,j]:=F[i,2*l-1,(j Div 2)]
                                  -F[i,2*l,(j Div 2)]
             else
              F[i+1,l,j]:=F[i,2*l-1,(j Div 2)]
                                  +F[i,2*l,(j Div 2)];

          End;

       End;
    End;

     For j:=0 to (N-1) Do
          Begin
             if isign=-1 then
               data[j]:=F[P,1,j]
             else
               data[j]:=(1/N)*F[P,1,j];

          End;
  End;

(*************************************************************)
(*            Procedure Prepare_input_block_for_Waltran      *)
(*                                                           *)
(*    This routine enters the data values into data[j] in    *)
(*    preparation for forward or inverse DWT.                *)
(*                                                           *)
(*      Called By:                                           *)
(*          Procedure Past_Current_Block                     *)
(*          Procedure Calculate_y                            *)
(*          Procedure Calculate_error                        *)
(*          Procedure Calculate_Gradient                     *)
(*          Procedure Update_weights                         *)
(*     Routines Called: None                                 *)
(*************************************************************)

Procedure Prepare_input_block_for_Waltran;

   Var
      j:integer;

         Begin

            if isign =1 then
```

```
                For j:=0 to (N-1) Do
                   begin
                      data[j]:=input_block[j];
                   end;
             if isign =-1 then
                For j:=0 to (N-1) Do
                   begin
                      data[j]:=V2[j];
                   end;

          End;


(*****************************************************************)
(*                 Procedure Past_Current_Block                  *)
(*                                                               *)
(*     This routine concatenates the current and previous blocks *)
(*     together: [(previous)(current)]. Each block is N points   *)
(*     long; the combination is 2*N points long.                 *)
(*                                                               *)
(*       Called By:                                              *)
(*          Main Program                                         *)
(*       Routines Called:                                        *)
(*          Procedure Prepare_input_block_for_Waltran            *)
(*          Procedure Waltran                                    *)
(*       Variables:                                              *)
(*           input_block[j] : (previous blck,current blck)       *)
(*****************************************************************)
(*****************************************************************)


Procedure Current_input_block;
  Var
    j:integer;

      Begin

          isign:=1;
          For j:=0 to (N-1) Do
           Begin
             input_block[j]:=input[j+k*N];
    (*  WriteLn(Lst, 'inputblock(', j,')=', input_block[j]); *)
           End;
          Prepare_input_block_for_Waltran;
          Waltran;
```

```
     End;
(*************************************************************)
(*                                                           *)
(*            Procedure Current_desired_block                *)
(*                                                           *)
(*************************************************************)
Procedure Current_desired_block;
  Var
    j:integer;
    Begin
        isign:=1;
        For j:=0 to (N-1) Do
           Begin
             input_block[j]:=desired[j+k*N];
           End;
        Prepare_input_block_for_Waltran;
        Waltran;
        For j:=0 to (N-1) Do
          Begin
            D[j]:=data[j];
        (*    WriteLn(Lst, '  D(',j,')= ',D[j]);  *)
          End;
    End;


(*************************************************************)
(*                    Procedure Load_input                   *)
(*                                                           *)
(*   This procedure reads in the input sequence from a data  *)
(*   file.                                                   *)
(*                                                           *)
(*       Called By:                                          *)
(*          Main Program                                     *)
(*       Routines Called: None                               *)
(*************************************************************)


Procedure Load_input;

  Var
     j:integer;

     Begin

        For j:=0 to datasize-1 Do
```

```
          ReadLn(infile,input[j]);

    End;

(*************************************************************)
(*                  Procedure Load_desired                   *)
(*                                                           *)
(*   This procedure reads in the desired sequence from a data*)
(*   file.                                                   *)
(*                                                           *)
(*      Called By:                                           *)
(*         Main Program                                      *)
(*      Routines Called: None                                *)
(*************************************************************)


Procedure Load_desired;
  Var
    j:integer;

     Begin

         For j:=0 to datasize-1 Do
             ReadLn(desiredfile,desired[j]);

     End;

(*************************************************************)
(*                  Procedure Write_output                   *)
(*                                                           *)
(*      This procedure writes the filter output and error    *)
(*      vectors to data files.                               *)
(*                                                           *)
(*        Called By:                                         *)
(*         Main Program                                      *)
(*        Routines Called:None                               *)
(*************************************************************)

Procedure Write_output;
    Var
       j:integer;

         Begin
            For j:=0 to datasize-1 Do
               Begin
```

```
                WriteLn(outfile,y[j]);
                WriteLn(errorfile,error[j]);
            End;

        End;

(*****************************************************************)
(*               Procedure Diagonal_of_X                         *)
(*                                                               *)
(*   This procedure creates the data vector X.                   *)
(*   The X data vector represents the diagonal component         *)
(*   of a diagonal matrix that contains the DWT of the           *)
(*   concatenated previous and current input blocks.             *)
(*                                                               *)
(*      Called By:                                               *)
(*          Main Program                                         *)
(*      Routines Called: None                                    *)
(*      Variables:                                               *)
(*        X[j] : diagonal  values                                *)
(*                                                               *)
(*****************************************************************)


Procedure Diagonal_of_X;
  Var
    j:integer;
    power:real;

        Begin

           For j:=0 to (N-1) Do
             begin

                 X[j]:=data[j];
        (*        WriteLn(Lst, '  X[',j, '] = ', X[j]); *)
             end;

        end;
(*****************************************************************)

Procedure Calculate_avg_input_bin_pwr;
   Var
      num,j:integer;
      Begin
        num:=Block_num-1;
```

```
        For k:=1 to Block_num-1 Do
          Begin
            Current_input_block;
            For j:=0 to (N-1) Do
              Pbin[j]:=Pbin[j]+Sqr(data[j]);
          End;
        For j:=0 to (N-1) Do
          Begin
            Pbin[j]:=(1/num)*Pbin[j];
     (*   WriteLn(Lst, ' Pbin(', j, ') =',Pbin[j]);  *)
          End;
      End;

(*****************************************************************)
Procedure Calculate_mu;
   Var
     j:integer;
     avgPwr:real;
     Begin
       avgPwr:=0;
       For j:=0 to (N-1) Do
        Begin
     (* gain_mu[j]:= Misadjust/(Pbin[j]+1.0E-8);  *)
         avgPwr:=avgPwr+Pbin[j]*(1/N);

        End;
     (*   gain_mu[6]:=gain_mu[7];   *)
        mu:=Misadjust/avgPwr;
 (*       WriteLn(Lst, '  gain constant mu = ', mu);   *)
 (*       WriteLn(Lst, ' average power = ',avgPwr);    *)

     End;

(*****************************************************************)

(*****************************************************************)
(*          Procedure Perform_Matrix_Multiply                    *)
(*                                                               *)
(*    Multiplies a 2Nx2N matrix by a 2Nx1 dimension vector.      *)
(*    The matrix in all cases is a diagonal matrix so the        *)
(*    routine automatically ignores the off diagonal terms       *)
(*    during the multiplication.                                 *)
(*                                                               *)
(*     Called By:                                                *)
(*       Procedure Calculate_y                                   *)
```

```
(*       Procedure Calculate_Gradient                                *)
(*     Routines Called: None                                         *)
(*     Variables:                                                    *)
(*       V2[j] : the resulting 2Nx1 vector                           *)
(*                                                                   *)
(*********************************************************************)

Procedure Perform_Matrix_Multiply(var M,V: RealArrayN2);


  Var
     j:integer;

       Begin
            For j:=0 to (N-1) Do
               Begin

                 V2[j]:=M[j]*V[j];
               End;

       End;

(*********************************************************************)
(*                Procedure Calculate_y                              *)
(*                                                                   *)
(*  This procedure calculates the output sequence values             *)
(*  for the current block being processed.  The output is            *)
(*  equal to the last N terms of the inverse DWT of the              *)
(*  product of X and the walsh domain weight vector.                 *)
(*                                                                   *)
(*     Called By:                                                    *)
(*       Main Program                                                *)
(*     Routines Called:                                              *)
(*       Procedure Waltran                                           *)
(*       Procedure Prepare_input_block_for_Waltran                   *)
(*       Procedure Perform_Matrix_Multiply                           *)
(*     Variables:                                                    *)
(*       y[j] : filter output                                        *)
(*                                                                   *)
(*********************************************************************)

Procedure Calculate_y;
  Var
   j:integer;
```

```
   Begin

     Perform_Matrix_Multiply(X,H);
     For j:=0 to (N-1) Do
        Yw[j]:=V2[j];
     isign:=-1;
     Prepare_input_block_for_Waltran;
     Waltran;
     For j:=0 to N-1 Do
       Begin
         y[k*N+j]:=data[j];
    (*     WriteLn(Lst, '  y(',k*N+j,')= ',y[k*N+j]); *)
       End;
   End;

(*****************************************************************)
(*                Procedure Calculate_error                      *)
(*                                                               *)
(*  This procedure calculates the error sequence values for      *)
(*  the current block and the complex error vector.              *)
(*  The error block for the current input                        *)
(*  block equals the current desired block minus the output      *)
(*  for the current input block.  The error vector               *)
(*  ERR[j], equals the FFT of the zero padded error block:       *)
(*  DWT[N zeros, error block].                                   *)
(*                                                               *)
(*    Called By:                                                 *)
(*      Main Program                                             *)
(*    Routines Called:                                           *)
(*      Procedure Prepare_input_block_for_Waltran                *)
(*      Procedure Waltran                                        *)
(*    Variables:                                                 *)
(*        ERR[j]  :   the DWT of the error sequence              *)
(*                  for the current block                        *)
(*                                                               *)
(*****************************************************************)


Procedure Calculate_error;
   Var
     j:integer;


      Begin
        For j:=0 to N-1 Do
```

```
          begin
              ERR[j]:=D[j]-Yw[j];
              V2[j]:=ERR[j];
          end;
        isign:=-1;
        Prepare_input_block_for_Waltran;
        Waltran;
        For j:=0 to (N-1) Do
            begin
               error[k*N+j]:=data[j];
            (*    WriteLn(Lst, '  e(',k*N+j,')=', error[k*N+j]); *)
            end;
    End;

(*****************************************************)
(*              Procedure Calculate_Gradient         *)
(*                                                   *)
(*  This procedure calculates the gradient sequence for the  *)
(*  current block being processed. The gradient sequence     *)
(*  equals the first N terms of the inverse DWTT of the      *)
(*  product of X and the error vector E.             *)
(*                                                   *)
(*   Called By:                                      *)
(*       Main Program                                *)
(*   Routines Called:                                *)
(*     Procedure Perform_Matrix_Multiply             *)
(*     Procedure Prepare_input_block_for_Waltran     *)
(*     Procedure Waltran                             *)
(*   Variables:                                      *)
(*     grad[j]:walsh-domain gradient vector for block k      *)
(*    Tgrad[j]: time-domain gradient vector for block k      *)
(*                                                   *)
(*****************************************************)


Procedure Calculate_Gradient;

  Var
     j:integer;
     Tgrad:RealArrayN2;

      Begin

          Perform_Matrix_Multiply(X,ERR);
          For j:=0 to (N-1) Do
```

```
        Begin
           grad[j]:=V2[j];
    (*  WriteLn(Lst, '  Grad(', j, ')= ', grad[j]);  *)
        End;
       isign:=-1;
       Prepare_input_block_for_Waltran;
       Waltran;

       For j:=0 to (N-1) Do
       Begin
         Tgrad[j]:=data[j];
    (*  WriteLn(Lst, '  grad(',j,')= ',Tgrad[j]); *)
       End;

    End;


(*****************************************************************)
(*                  Procedure Update_weights                     *)
(*                                                               *)
(*   This procedure updates the filters tap weights.  The        *)
(*   new weights equal the old weights plus the product of the   *)
(*   gain constant and the gradient vector.  The                 *)
(*   gradient vector equals the DWT of the gradient              *)
(*   sequence padded with N zeros: DWT[(grad seq), N zeros].     *)
(*                                                               *)
(*     Called By:                                                *)
(*        Main Program                                           *)
(*     Routines Called:                                          *)
(*       Procedure Prepare_input_block_for_Waltran               *)
(*       Procedure Waltran                                       *)
(*     Variables:                                                *)
(*       H[j] : Walsh domain weight vector                       *)
(*****************************************************************)


Procedure Update_weights;

    Var
       tap,j:integer;

        Begin
          For tap:=0 to (N-1) Do
              begin
                 H[tap]:=H[tap]+mu*grad[tap];
```

```
        (*       WriteLn(Lst, '  H(', tap, ')= ', H[tap]); *)

              end;
         End;
(*************************************************************)
(*             Procedure time_domain_wts                     *)
(*************************************************************)
Procedure time_domain_wts;
  Var
    j:integer;
    Begin
       For j:=0 to (N-1) Do
          Begin
            V2[j]:=H[j];
    (*     WriteLn(Lst, 'H(',j,')=', H[j]);   *)
          End;
       isign:=-1;
       Prepare_input_block_for_Waltran;
       Waltran;
    (*   WriteLn(Lst, ' Block ', k);    *)
       For j:=0 to (N-1) Do
          Begin
            W[j]:=data[j];
     (*     WriteLn(Lst, '  W(', j, ') =', W[j]); *)
          End;
    End;

(*************************************************************)
(*                    Procedure Set_weights                  *)
(*************************************************************)

Procedure Set_weights;
   Var
     j:integer;

     Begin
       W[0]:=48.4796;
       W[1]:=5.6844;
       W[2]:=5.0732;
       W[3]:=10.7032;
       For j:=0 to (N-1) Do
         input_block[j]:=W[j];
       isign:=1;
       Prepare_input_block_for_Waltran;
       Waltran;
```

```
      For j:=0 to (N-1) Do
         H[j]:=data[j];
    End;
(*****************************************************************)
(************************  Main Program  *************************)
(*****************************************************************)


Begin
   Open_input_files;
   Open_output_files;
   Init_var;
   Load_input;
   Load_desired;
(*   ClockOn;   *)
   Calc_numblocks;
   Calculate_avg_input_bin_pwr;
   Calculate_mu;
(*   Set_weights;  *)
   For k:=0 to Block_num-1 Do
     Begin
      WriteLn( '  processing block',  k);
      Current_input_block;
      Diagonal_of_X;
      Current_desired_block;
      Calculate_y;
      Calculate_error;
      Calculate_gradient;
      Update_weights;
  (*    time_domain_wts; *)
     end;
(*   time_domain_wts;  *)
   Write_output;
   Close_input_files;
   Close_output_files;

End.
```

F.2 *WDF2 Filter Listing*

This is the Turbo Pascal 6.0 listing of the WDF2 program.

```
Program WDF2;
{$N+}
Uses Printer,CRT,DOS;

CONST
  N=8;          (*** N is block size*)
  datasize=1000;
  Misadjust=0.1;
  P=4;          (* 2N=2**P *)
TYPE

  RealArrayN2=ARRAY[0..(2*N)-1] of real;
  InputArray=ARRAY[0..datasize-1] of real;
  OutputArray=ARRAY[0..datasize-1] of real;
  FArray=ARRAY[0..P,1..(2*N),0..(2*N-1)] of Real;
VAR
  hr,m,s,s100:Word;
  StartClock,StopClock:Real;
  data:RealArrayN2;
  F:FArray;
  gain_mu,Pbin,X,H,W,ERR,V2,input_block,grad:RealArrayN2;
  desired,input:InputArray;
  error,y:OutputArray;
  isign,nn,Block_num,k:integer;
  mu:real;
  infile,desiredfile,errorfile,outfile,weight0:text;
  weight1,weight2,weight3,weight4,weight5,weight6,weight7:text;
 (*************************************************************)
 (*                   Procedure Init_var                      *)
 (*                                                           *)
 (*    This procedure initializes all variables.              *)
 (*                                                           *)
 (*     Called By:                                            *)
 (*         Main Program                                      *)
 (*     Routines Called: None                                 *)
 (*************************************************************)

 Procedure Init_var;
   Var
     l,j,i:integer;
```

```
    Begin

      Block_num:=0;
      For j:=0 to datasize-1 Do
        Begin
          error[j]:=0;
          desired[j]:=0;
          input[j]:=0;
          y[j]:=0;
        End;
      For j:=0 to (2*N)-1 Do
         Begin
           X[j]:=0;
           W[j]:=0;
           H[j]:=0;
           ERR[j]:=0;
           V2[j]:=0;
           input_block[j]:=0;
           grad[j]:=0;
           gain_mu[j]:=0;
           Pbin[j]:=0;
        End;
       For j:=0 to P Do
         For i:=1 to (2*N) Do
          For l:=0 to (2*N-1) Do
            F[j,i,l]:=0;
    End;

(*************************************************************)
(*                Procedure Open input files                 *)
(*                                                           *)
(*    Called By:                                             *)
(*         Main Program                                      *)
(*    Routines Called: None                                  *)
(*************************************************************)
Procedure Open_input_files;
   Begin
      Assign(infile, 'B:tstinpt2.Dat');
      Assign(desiredfile, 'B:tstdes2.Dat');
      Reset(infile);
      Reset(desiredfile);
   End;

(*************************************************************)
```

```
(*                    Procedure Open output files                *)
(*                                                               *)
(*      Called By:                                               *)
(*         Main Program                                          *)
(*      Routines Called: None                                    *)
(***************************************************************)
Procedure Open_output_files;
   Begin
      Assign(errorfile, 'B:error.Dat');
      Rewrite(errorfile);
      Assign(outfile, 'B:out.dat');
      Rewrite(outfile);
   End;

(***************************************************************)
(*                  Procedure Close input files                *)
(*                                                             *)
(*     Called By:                                              *)
(*         Main Program                                        *)
(*     Routines Called: None                                   *)
(*************************************************************)
Procedure Close_input_files;
   Begin
      Close(infile);
      Close(desiredfile);
   End;

(*************************************************************)
(*                Procedure Close output files               *)
(*                                                           *)
(*      Called By:                                           *)
(*          Main Program                                     *)
(*      Routines Called: None                                *)
(***********************************************************)
Procedure Close_output_files;
   Begin
      Close(errorfile);
      Close(outfile);

   End;

(***********************************************************)
(*                  Procedure Calc_numblocks               *)
(*                                                         *)
(*     This procedure calculates the number of blocks to   *)
```

```
(*     be processed.                                          *)
(*                                                            *)
(*        Called By:                                          *)
(*           Main Program                                     *)
(*       Routines Called: None                                *)
(************************************************************)

Procedure Calc_numblocks;
  Begin

   Block_num:=datasize div N;

  End;


 (*****************************************************************)
 (*                  Procedure Waltran                            *)
 (*                                                               *)
 (*   This procedure performs the forward and inverse 2N-point    *)
 (*   Fast Walsh Transform (FWT).    This routine calculates      *)
 (*   the forward and inverse transforms using the same loop:     *)
 (*   the forward transform requires the loop result to be        *)
 (*   multiplied by a factor of 1/(2N).                           *)
 (*   The algorithm is recursive and requires P stages, where     *)
 (*   2N=2**P.  This routine requires no multiplications          *)
 (*   and 2NLog(2N) summations, where the logarithm is base 2.    *)
 (*                                                               *)
 (*    Called By:                                                 *)
 (*      Procedure Past_Current_Block                             *)
 (*      Procedure Calculate_y                                    *)
 (*      Procedure  Calculate_error                               *)
 (*      Procedure  Calculate_Gradient                            *)
 (*      Procedure Update_Weights                                 *)
 (*    Routines Called:                                           *)
 (*        None                                                   *)
 (*    Variables:                                                 *)
 (*           P: 2N=2**P                                          *)
 (*      isign : indicates inverse or forward transform           *)
 (*      data[j] : input and output                               *)
 (*      F[P,1,j] :Fast Walsh Matrix result                       *)
 (*            j=0,1,..2N-1                                       *)
 (*****************************************************************)
 (*****************************************************************)
 Procedure Waltran;
```

```
Var
   j,l,i,jmax,lmax,exponent:integer;
   jlog,jinv:real;

   Begin
    For l:=1 to (2*N) Do
      Begin
        F[0,l,0]:=data[l-1];
      (*  WriteLn(Lst, '  F(',0,l,0,')= ',F[0,l,0]); *)
      End;
    For i:=0 to P-1 Do
     Begin
      jlog:=(i+1)*Ln(2);
      jinv:=Exp(jlog);
      jmax:=Round(jinv);
 (*     WriteLn(Lst, 'jmax= ', jmax); *)
      lmax:=(2*N) Div jmax;
  (*    WriteLn(Lst, ' kmax= ', kmax); *)
      For l:=1 to lmax Do
        Begin
         For j:=0 to jmax-1 Do
           Begin
            exponent:=(j+1) Div 2;
            If Odd(exponent) then
              F[i+1,l,j]:=F[i,2*l-1,(j Div 2)]
                                  -F[i,2*l,(j Div 2)]
             else
              F[i+1,l,j]:=F[i,2*l-1,(j Div 2)]
                                  +F[i,2*l,(j Div 2)];

           End;

        End;
     End;

      For j:=0 to ((2*N)-1) Do
           Begin
              if isign=-1 then
                data[j]:=F[P,1,j]
              else
                data[j]:=(1/(2*N))*F[P,1,j];

           End;

    End;
```

```
(*****************************************************************)
(*             Procedure Prepare_input_block_for_Waltran          *)
(*                                                                *)
(*    This routine enters the data values into data[j] in         *)
(*    preparation for forward or inverse DWT.                     *)
(*                                                                *)
(*      Called By:                                                *)
(*          Procedure Past_Current_Block                          *)
(*          Procedure Calculate_y                                 *)
(*          Procedure Calculate_error                             *)
(*          Procedure Calculate_Gradient                          *)
(*          Procedure Update_weights                              *)
(*     Routines Called: None                                      *)
(*****************************************************************)

Procedure Prepare_input_block_for_Waltran;

   Var
      j:integer;

         Begin

            if isign =1 then
                For j:=0 to (2*N)-1 Do
                   begin
                      data[j]:=input_block[j];
                   end;
            if isign =-1 then
                For j:=0 to (2*N)-1 Do
                   begin
                      data[j]:=V2[j];
                   end;

         End;


(*****************************************************************)
(*                Procedure Past_Current_Block                   *)
(*                                                               *)
(*     This routine concatenates the current and previous blocks *)
(*     together: [(previous)(current)]. Each block is N points   *)
(*     long; the combination is 2*N points long.                 *)
(*                                                               *)
```

```
(*       Called By:                                                    *)
(*          Main Program                                               *)
(*       Routines Called:                                              *)
(*          Procedure Prepare_input_block_for_Waltran                  *)
(*          Procedure Waltran                                          *)
(*       Variables:                                                    *)
(*           input_block[j] : (previous blck,current blck)             *)
(*****************************************************************)
(*****************************************************************)

Procedure Past_Current_Block;
  Var
    j:integer;

      Begin

          isign:=1;
          For j:=0 to (2*N)-1 Do
           Begin
            if (k*N-N+j<0) then
               input_block[j]:=0
             else
               input_block[j]:=input[j+k*N-N];

           End;
          Prepare_input_block_for_Waltran;
          Waltran;

      End;

(*****************************************************************)
(*                    Procedure Load_input                       *)
(*                                                               *)
(*   This procedure reads in the input sequence from a data      *)
(*   file.                                                       *)
(*                                                               *)
(*       Called By:                                              *)
(*          Main Program                                         *)
(*       Routines Called: None                                   *)
(*****************************************************************)

Procedure Load_input;

  Var
     j:integer;
```

```
    Begin

       For j:=0 to datasize-1 Do
          ReadLn(infile,input[j]);

    End;



(*****************************************************************)
(*                  Procedure Load_desired                       *)
(*                                                               *)
(*   This procedure reads in the desired sequence from a data    *)
(*   file.                                                       *)
(*                                                               *)
(*      Called By:                                               *)
(*         Main Program                                          *)
(*      Routines Called: None                                    *)
(*****************************************************************)

Procedure Load_desired;
  Var
    j:integer;

      Begin

          For j:=0 to datasize-1 Do
              ReadLn(desiredfile,desired[j]);

      End;

(*****************************************************************)
(*                  Procedure Write_output                       *)
(*                                                               *)
(*      This procedure writes the filter output and error        *)
(*      vectors to data files.                                   *)
(*                                                               *)
(*        Called By:                                             *)
(*         Main Program                                          *)
(*        Routines Called:None                                   *)
(*****************************************************************)

Procedure Write_output;
    Var
```

```
      j:integer;

         Begin
            For j:=0 to datasize-1 Do
               Begin
                  WriteLn(outfile,y[j]);
                  WriteLn(errorfile,error[j]);
               End;

         End;
  (*****************************************************)
 Procedure ClockOn;
 Begin
   GetTime(hr,m,s,s100);
   StartClock:=(hr*3600)+(m*60)+s+(s100/100);
 End;
(*************************************************************)
Procedure ClockOff;
 Begin
   GetTime(hr,m,s,s100);
   StopClock:=(hr*3600)+(m*60)+s+(s100/100);
   WriteLn('Elapsed time = ',(StopClock-StartClock):0:2);
 End;

(*************************************************************)
(*                Procedure Diagonal_of_X                    *)
(*                                                           *)
(*   This procedure creates the data vector X.               *)
(*   The X data vector represents the diagonal component     *)
(*   of a diagonal matrix that contains the DWT of the       *)
(*   concatenated previous and current input blocks.         *)
(*                                                           *)
(*      Called By:                                           *)
(*          Main Program                                     *)
(*      Routines Called: None                                *)
(*      Variables:                                           *)
(*        X[j] : diagonal  values                            *)
(*                                                           *)
(*************************************************************)

Procedure Diagonal_of_X;
  Var
    j:integer;
    power:real;
```

```
        Begin

           For j:=0 to (2*N)-1 Do
             begin

                 X[j]:=data[j];
            (*    WriteLn(Lst, '  X[',j, '] = ',X[j]); *)
             end;

        end;
(*****************************************************************)

Procedure Calculate_avg_input_bin_pwr;
   Var
      num,j:integer;
      Begin
        num:=Block_num-1;
        For k:=1 to Block_num-1 Do
          Begin
            Past_current_block;

            For j:=0 to (2*N)-1 Do
              Begin
               Pbin[j]:=Pbin[j]+Sqr(data[j]);
              End;
          End;
        For j:=0 to (2*N)-1 Do
          Begin
            Pbin[j]:=(1/num)*Pbin[j];
    (*    WriteLn(Lst, ' Pbin(', j, ') =',Pbin[j]); *)
          End;
       End;
(*************************************************************)

(*****************************************************************)
Procedure Calculate_mu;
   Var
     j:integer;
     avgPwr:real;
     Begin
       avgPwr:=0;
       For j:=0 to (2*N)-1 Do
        Begin
         avgPwr:=avgPwr+Pbin[j]*(1/(2*N));
     (*    gain_mu[j]:= Misadjust/(Pbin[j]+1.0E-8);   *)
```

```
        End;
    (*   gain_mu[6]:=gain_mu[7];   *)
        mu:=Misadjust/avgPwr;
        WriteLn(Lst, '  gain constant mu = ', mu);
  (*      WriteLn(Lst, ' average power = ', avgPwr);   *)
    End;
(*************************************************************)

(*************************************************************)
(*          Procedure Perform_Matrix_Multiply                *)
(*                                                           *)
(*    Multiplies a 2Nx2N matrix by a 2Nx1 dimension vector.  *)
(*    The matrix in all cases is a diagonal matrix so the    *)
(*    routine automatically ignores the off diagonal terms   *)
(*    during the multiplication.                             *)
(*                                                           *)
(*     Called By:                                            *)
(*       Procedure Calculate_y                               *)
(*       Procedure Calculate_Gradient                        *)
(*     Routines Called: None                                 *)
(*     Variables:                                            *)
(*       V2[j] : the resulting 2Nx1 vector                   *)
(*                                                           *)
(*************************************************************)

Procedure Perform_Matrix_Multiply(var M,V: RealArrayN2);


  Var
     j:integer;

       Begin

           For j:=0 to (2*N)-1 Do
               Begin
                 V2[j]:=M[j]*V[j];
               End;
       End;

(*************************************************************)
(*                Procedure Calculate_y                      *)
(*                                                           *)
(*  This procedure calculates the output sequence values     *)
(*  for the current block being processed.  The output is    *)
```

```
(*  equal to the last N terms of the inverse DWT of the       *)
(*  product of X and the walsh domain weight vector.          *)
(*                                                            *)
(*     Called By:                                             *)
(*       Main Program                                         *)
(*     Routines Called:                                       *)
(*       Procedure Waltran                                    *)
(*       Procedure Prepare_input_block_for_Waltran            *)
(*       Procedure Perform_Matrix_Multiply                    *)
(*     Variables:                                             *)
(*       y[j] : filter output                                 *)
(*                                                            *)
(************************************************************)

Procedure Calculate_y;
  Var
   j:integer;

   Begin

     Perform_Matrix_Multiply(X,H);
     isign:=-1;
     Prepare_input_block_for_Waltran;
     Waltran;
     For j:=0 to N-1 Do
       Begin
         y[k*N+j]:=data[N+j];
     (*    WriteLn(Lst, '  y(',k*N+j,')= ',y[k*N+j]);   *)
       End;

   End;

(************************************************************)
(*                Procedure Calculate_error                 *)
(*                                                          *)
(*  This procedure calculates the error sequence values for *)
(*  the current block and the complex error vector.         *)
(*  The error block for the current input                   *)
(*  block equals the current desired block minus the output *)
(*  for the current input block.  The error vector          *)
(*  ERR[j], equals the FFT of the zero padded error block:  *)
(*  DWT[N zeros, error block].                              *)
(*                                                          *)
(*    Called By:                                            *)
(*      Main Program                                        *)
```

```
(*    Routines Called:                                          *)
(*      Procedure Prepare_input_block_for_Waltran               *)
(*      Procedure Waltran                                       *)
(*    Variables:                                                *)
(*        ERR[j]  :   the DWT of the error sequence             *)
(*                  for the current block                       *)
(*                                                              *)
(****************************************************************)

Procedure Calculate_error;
   Var
     j:integer;
     e:RealArrayN2;

      Begin

        For j:=0 to N-1 Do
           begin
               e[j]:=0;
               input_block[j]:=e[j];
           end;
        For j:=N to (2*N)-1 Do
            Begin
               e[j]:=desired[k*N+j-N]-y[k*N+j-N];
               input_block[j]:=e[j];
               error[k*N+j-N]:=e[j];
    (*     WriteLn(Lst, '  e(',k*N+j-N,')=', error[k*N+j-N]);  *)
            End;
        isign:=1;
        Prepare_input_block_for_Waltran;
        Waltran;
        For j:=0 to (2*N)-1 Do
            begin
               ERR[j]:=data[j];

            end;
     End;

(****************************************************************)
(*              Procedure Calculate_Gradient                    *)
(*                                                              *)
(*  This procedure calculates the gradient sequence for the     *)
(*  current block being processed. The gradient sequence        *)
(*  equals the first N terms of the inverse DWTT of the         *)
(*  product of X and the error vector E.                        *)
```

```
(*                                                                  *)
(*   Called By:                                                     *)
(*       Main Program                                               *)
(*   Routines Called:                                               *)
(*     Procedure Perform_Matrix_Multiply                            *)
(*     Procedure Prepare_input_block_for_Waltran                    *)
(*     Procedure Waltran                                            *)
(*   Variables:                                                     *)
(*     grad[j]: gradient sequence for current block                 *)
(*                                                                  *)
(********************************************************************)

Procedure Calculate_Gradient;

  Var
     j:integer;
     Tgrad:RealArrayN2;

      Begin

          Perform_Matrix_Multiply(X,ERR);
          For j:=0 to (2*N)-1 Do
           Begin
              grad[j]:=V2[j];
 (*           WriteLn(Lst, '  Grad(', j, ')= ', grad[j]);  *)
           End;
    (*       isign:=-1;                          *)
    (*      Prepare_input_block_for_Waltran;  *)
    (*      Waltran;                  *)

    (*      For j:=0 to (2*N)-1 Do  *)   (* Calculate the time *)
    (*      Begin                 *)     (* domain gradient *)
    (*        Tgrad[j]:=data[j];       *)
   (*       WriteLn(Lst, '  grad(',j,')= ',Tgrad[j]);   *)
    (*      End;                      *)

      End;


(********************************************************************)
(*                 Procedure Update_weights                         *)
(*                                                                  *)
(*   This procedure updates the filters tap weights.  The           *)
(*   new weights equal the old weights plus the product of the      *)
(*   gain constant and the gradient vector.  The                    *)
```

```
(*   gradient vector equals the DWT of the gradient            *)
(*   sequence padded with N zeros: DWT[(grad seq), N zeros].   *)
(*                                                             *)
(*     Called By:                                              *)
(*        Main Program                                         *)
(*     Routines Called:                                        *)
(*       Procedure Prepare_input_block_for_Waltran             *)
(*       Procedure Waltran                                     *)
(*     Variables:                                              *)
(*       H[j] : Walsh domain weight vector                     *)
(*************************************************************)

Procedure Update_weights;

     Var
        tap,j:integer;

          Begin

           For tap:=0 to (2*N)-1 Do
                begin
                   H[tap]:=H[tap]+mu*grad[tap];
           (*  WriteLn(Lst, ' H(', tap, ')= ', H[tap]); *)

                end;

          End;
(*************************************************************)
(*             Procedure time_domain_wts                     *)
(*************************************************************)
Procedure time_domain_wts;
  Var
    j:integer;
    Begin
       For j:=0 to (2*N)-1 Do
          Begin
            V2[j]:=H[j];
  (*    WriteLn(Lst, 'H(',j,')=', H[j]);    *)
          End;
       isign:=-1;
       Prepare_input_block_for_Waltran;
       Waltran;
    (*   WriteLn(Lst, ' Block ', k);    *)
       For j:=0 to (2*N)-1 Do
          Begin
```

```
            W[j]:=data[j];

          End;
      End;

(*****************************************************  *****)
(*                      Procedure Set_weights                 *)
(*************************************************************)

Procedure Set_weights;
   Var
     j:integer;

     Begin
       W[0]:=48.4796;
       W[1]:=5.6844;
       W[2]:=5.0732;
       W[3]:=10.7032;
       For j:=0 to (2*N)-1 Do
         input_block[j]:=W[j];
       isign:=1;
       Prepare_input_block_for_Waltran;
       Waltran;
       For j:=0 to (2*N)-1 Do
          H[j]:=data[j];
     End;
(*************************************************************)
(************************  Main Program  *********************)
(*************************************************************)

  Begin
    Open_input_files;
    Open_output_files;
    Init_var;
    Load_input;
    Load_desired;

    Calc_numblocks;
    Calculate_avg_input_bin_pwr;
     Calculate_mu;
    (*   Set_weights;  *)
    (*   ClockOn; *)        (* Turn Clock on *)
    For k:=0 to Block_num-1 Do
      Begin
   (*    WriteLn( '  processing block',  k);  *)
```

```
        Past_current_block;
        Diagonal_of_X;
        Calculate_y;
        Calculate_error;
        Calculate_gradient;
        Update_weights;
      (*  time_domain_wts;    *)
       end;
  (*    ClockOff; *)  (*Display elapsed time*)
  (*    WriteLn;  *)
  (*    Write('Press Enter.. ');*)
  (*    ReadLn;   *)
(*    time_domain_wts;    *)
      Write_output;
      Close_input_files;
      Close_output_files;

   End.
```

F.3 FDF Filter Listing

This is the Turbo Pascal 6.0 listing of the FDF program.

```
Program Freq_adfil;
    {$N+}
    Uses Printer,CRT,DOS;
    CONST
      N=16;          (*** N is block size*)
      datasize=992;
      Misadjust=0.1;

    TYPE
      RealArrayN4=ARRAY[1..4*N] of real;
      RealArrayN2=ARRAY[1..2*N] of real;
      InputArray=ARRAY[1..datasize] of real;
      OutputArray=ARRAY[1..datasize] of real;

    VAR
      hrr,m,s,s100:Word;
      StopClock,StartClock:Real;
      data:RealArrayN4;
      XR,XI,XI_conj,HR,ER,WR,WI,EI,HI,V2R,V2I,input_block,grad:RealArrayN2;
      igain_mu,rgain_mu,Pbinr,Pbini:RealArrayN2;
      desired,input:InputArray;
      error,y:OutputArray;
      isign,nn,Block_num,k:integer;
      mu:real;
      infile,desiredfile,errorfile,outfile:text;
      weight0,weight1,weight2,weight3,weight4,weight5,weight6,weight7:text;
     (*************************************************************)
     (*                    Procedure Init_var                     *)
     (*                                                           *)
     (*    This procedure initializes all variables.              *)
     (*                                                           *)
     (*     Called By:                                            *)
     (*         Main Program                                      *)
     (*     Routines Called: None                                 *)
     (*************************************************************)

      Procedure Init_var;
        Var
          j:integer;
```

```
    Begin

      Block_num:=0;
      For j:=1 to datasize Do
        Begin
          error[j]:=0;
          desired[j]:=0;
          input[j]:=0;
          y[j]:=0;
        End;
      For j:=1 to 2*N Do
         Begin
           XR[j]:=0;
           XI[j]:=0;
           XI_conj[j]:=0;
           HR[j]:=0;
           HI[j]:=0;
           ER[j]:=0;
           EI[j]:=0;
           V2R[j]:=0;
           V2I[j]:=0;
           input_block[j]:=0;
           grad[j]:=0;
           igain_mu[j]:=0;
           rgain_mu[j]:=0;
           Pbinr[j]:=0;
           Pbini[j]:=0;
        End;

    End;
(*************************************************************)
Procedure Open_input_files;
   Begin
     Assign(infile, 'A:S53N.Dat');
     Assign(desiredfile, 'A:S5.Dat');
     Reset(infile);
     Reset(desiredfile);
   End;

(*************************************************************)
Procedure Open_output_files;
   Begin
     Assign(errorfile, 'B:FFS3SNE.Dat');
     Rewrite(errorfile);
     Assign(outfile, 'B:FFS4SNY.Dat');
```

```
    Rewrite(outfile);

  End;

(*********************************************************)
Procedure Close_input_files;
  Begin
    Close(infile);
    Close(desiredfile);
  End;
(*************************************************************)
Procedure Close_output_files;
  Begin
    Close(errorfile);
    Close(outfile);

  End;
(*************************************************************)
(*                  Procedure Calc_numblocks                 *)
(*                                                           *)
(*     This procedure calculates the number of blocks to     *)
(*     be processed.                                         *)
(*                                                           *)
(*        Called By:                                         *)
(*           Main Program                                    *)
(*       Routines Called: None                               *)
(*************************************************************)

Procedure Calc_numblocks;
  Begin

   Block_num:=datasize div N;

  End;


(*************************************************************)
(*                     Procedure FFT                         *)
(*                                                           *)
(*    This procedure calculates the forward and inverse Fast *)
(*    Fourier Transform of a data sequence that has a power of *)
(*    2 number of data points.  Both real and complex data can *)
(*    be transformed. The routine has two sections. The first *)
(*    section sorts the input data into bit-reversed order. The *)
(*    second section has an outer loop that is executed log N *)
```

```
(*    (power 2) times. Transforms of length 2,4,8,...,N are       *)
(*    calculated in this section.  The section has two nested     *)
(*    inner loops that execute the Danielson-Lanczos Lemma.       *)
(*    Data tranformed or inverse transformed is entered into      *)
(*    the vector data[j] according to R1,I1,R2,I2,...RN,IN;       *)
(*    where R1 and I1 represent the real and imaginary            *)
(*    components of the first data value or transform value.      *)
(*  Transform results are returned in data[j] in the same fashion.*)
(*    The forward and inverse transforms are indicated to the     *)
(*    routine by setting the isign flag: isign=1 indicates        *)
(*    forward transform, isign=-1 indicates inverse transform.    *)
(*    In the case of inverse transform the resultant              *)
(*    is scaled by a factor equal to the number of points         *)
(*    transformed.                                                *)
(*                                                                *)
(*       Called By:                                               *)
(*           Procedure Past_Current_Block                         *)
(*           Procedure  Calculate_y                               *)
(*           Procedure  Calculate_error                           *)
(*           Procedure  Calculate_Gradient                        *)
(*           Procedure Update_weights                             *)
(*       Routines Called: None                                    *)
(*       Variables:                                               *)
(*           isign: indicates inverse or forward transform        *)
(*           data[j]: input and output                            *)
(*           nn: number of points to be transformed               *)
(*           n  : number of Re and Im value in data[j] =2*nn      *)
(*****************************************************************)
Procedure FFT;


  VAR
    ii,jj,n,mmax,m,j,istep,i:integer;
    wtemp,wr,wpr,wpi,wi,theta:double;
    tempr,tempi,wrs,wis:real;
 Begin
    n:=64;   (* 2 times 2N *)
    nn:=32;  (* 2N *)
    j:=1;

    For ii:=1 To nn Do Begin

       i:=2*ii-1;
       If j>i Then Begin
          tempr:=data[j];
```

```
        tempi:=data[j+1];
        data[j]:=data[i];
        data[j+1]:=data[i+1];
        data[i]:=tempr;
        data[i+1]:=tempi;
    End;
    m:=n DIV 2;
    While (m >= 2) And (j > m) Do Begin
          j:=j-m;
          m:=m DIV 2
    End;
    j:=j+m;

End;

mmax:=2;
While n > mmax Do Begin

      istep:=2*mmax;
      theta:=6.28318530717959/(isign*mmax);
      wpr:=-2.0*sqr(sin(0.5*theta));
      wpi:=sin(theta);
      wr:=1.0;
      wi:=0.0;
      For ii:=1 To mmax DIV 2 Do Begin

          m:=2*ii-1;
          wrs:=wr;
          wis:=wi;
          For jj:=0 To (n-m) DIV istep Do Begin

              i:=m+jj*istep;
              j:=i+mmax;
              tempr:=wrs*data[j]-wis*data[j+1];
              tempi:=wrs*data[j+1]+wis*data[j];
              data[j]:=data[i]-tempr;
              data[j+1]:=data[i+1]-tempi;
              data[i]:=data[i]+tempr;
              data[i+1]:=data[i+1]+tempi

          End;

          wtemp:=wr;
          wr:=wr*wpr-wi*wpi+wr;
          wi:=wi*wpr+wtemp*wpi+wi;
```

```
        End;

        mmax:=istep;
    End;

End;
```

```
(*****************************************************************)
(*             Procedure Prepare_input_block_for_FFT             *)
(*                                                               *)
(*    This routine enters the data values into data[j] in        *)
(*    preparation for forward or inverse FFT.                    *)
(*                                                               *)
(*      Called By:                                               *)
(*          Procedure Past_Current_Block                         *)
(*          Procedure Calculate_y                                *)
(*          Procedure Calculate_error                            *)
(*          Procedure Calculate_Gradient                         *)
(*          Procedure Update_weights                             *)
(*     Routines Called: None                                     *)
(*****************************************************************)

Procedure Prepare_input_block_for_FFT;

   Var
      j:integer;

         Begin

            if isign =1 then
                For j:=1 to 2*N Do
                   begin
                      data[2*j-1]:=input_block[j];
                      data[2*j]:=0;
                   end;
            if isign =-1 then
                For j:=1 to 2*N Do
                   begin
                      data[2*j-1]:=V2R[j];
                      data[2*j]:=V2I[j];
                   end;
```

```
        End;


(*****************************************************************)
(*                 Procedure Past_Current_Block                  *)
(*                                                               *)
(*     This routine concatenates the current and previous blocks *)
(*     together: [(previous)(current)]. Each block is N points   *)
(*     long; the combination is 2*N points long.                 *)
(*                                                               *)
(*       Called By:                                              *)
(*          Main Program                                         *)
(*       Routines Called:                                        *)
(*          Procedure Prepare_input_block_for_FFT                *)
(*          Procedure FFT                                        *)
(*       Variables:                                              *)
(*           input_block[j] : (previous blck,current blck)       *)
(*****************************************************************)
(*****************************************************************)


Procedure Past_Current_Block;
  Var
    j:integer;

      Begin

          isign:=1;
          For j:=1 to 2*N Do
            if (k*N-N+j-1<0) then
               input_block[j]:=0
             else
               input_block[j]:=input[j+k*N-N];
          Prepare_input_block_for_FFT;
          FFT;

      End;


(*****************************************************************)
(*                    Procedure Load_input                       *)
(*                                                               *)
(*   This procedure reads in the input sequence from a data      *)
(*   file.                                                       *)
(*                                                               *)
```

```
(*       Called By:                                                    *)
(*          Main Program                                               *)
(*       Routines Called: None                                         *)
(*******************************************************************)


Procedure Load_input;

  Var
     j:integer;

     Begin

        For j:=1 to datasize Do
           ReadLn(infile,input[j]);

     End;



(*******************************************************************)
(*                  Procedure Load_desired                         *)
(*                                                                 *)
(*   This procedure reads in the desired sequence from a data      *)
(*   file.                                                         *)
(*                                                                 *)
(*      Called By:                                                 *)
(*         Main Program                                            *)
(*      Routines Called: None                                      *)
(*******************************************************************)


Procedure Load_desired;
  Var
    j:integer;

      Begin

          For j:=1 to datasize Do
              ReadLn(desiredfile,desired[j]);

      End;

(*******************************************************)
(*                  Procedure Write_output             *)
```

```
(*                                                          *)
(*      This procedure writes the filter output and error   *)
(*      vectors to data files.                              *)
(*                                                          *)
(*        Called By:                                        *)
(*         Main Program                                     *)
(*        Routines Called:None                              *)
(************************************************************)

Procedure Write_output;
    Var
       j:integer;

          Begin
             For j:=1 to datasize Do
                Begin
                    WriteLn(outfile,y[j]);
                    WriteLn(errorfile,error[j]);
                End;

          End;
 (************************************************************)
 Procedure ClockOn;
 Begin
   GetTime(hrr,m,s,s100);
   StartClock:=(hrr*3600)+(m*60)+s+(s100/100);
 End;
(************************************************************)
Procedure ClockOff;
 Begin
   GetTime(hrr,m,s,s100);
   StopClock:=(hrr*3600)+(m*60)+s+(s100/100);
   WriteLn('Elapsed time = ',(StopClock-StartClock):0:2);
 End;

(************************************************************)
(*                Procedure Diagonal_of_X                   *)
(*                                                          *)
(*   This procedure creates two data vectors: XR and XI.    *)
(*   The two data vectors represent the diagonal component  *)
(*   of a diagonal matrix that contains the FFT of the      *)
(*   concatenated previous and current input blocks. XR     *)
(*   represents the real part of each value and XI represents *)
(*   the corresponding imaginary part:(diagonal of X)=XR+XI. *)
(*                                                          *)
```

```
(*      Called By:                                                *)
(*          Main Program                                          *)
(*      Routines Called: None                                     *)
(*      Variables:                                                *)
(*        XR[j] : diagonal real values                            *)
(*        XI[j] : diagonal imaginary values                       *)
(*                                                                *)
(******************************************************************)


Procedure Diagonal_of_X;
  Var
    j:integer;
    power:real;

        Begin

           For j:=1 to 2*N Do
             begin
               XR[j]:=data[2*j-1];
               XI[j]:=data[2*j];
            WriteLn(Lst, ' X(',j-1,')= ',XR[j],'+i',XI[j]);

             end;

        end;
(******************************************************************)
Procedure Calculate_avg_input_bin_pwr;
   Var
     num,j:integer;
     Begin
       num:=Block_num-1;
       For k:=1 to Block_num-1 Do
         Begin
           Past_current_block;
           For j:=1 to 2*N Do
             Begin
               Pbinr[j]:=Pbinr[j]+Sqr(data[2*j-1]);
               Pbini[j]:=Pbini[j]+Sqr(data[2*j]);
             End;
         End;
       For j:=1 to 2*N Do
          Begin
            Pbinr[j]:=(1/num)*Pbinr[j];
            Pbini[j]:=(1/num)*Pbini[j];
```

```
        End;
    End;

(*************************************************************)
Procedure Calculate_mu;
  Var
    j:integer;
    avgPwr,ravgPwr,iavgPwr:Real;
    Begin
      avgPwr:=0.0;
      ravgPwr:=0;
      iavgPwr:=0;
      For j:=1 to 2*N Do
        Begin
          ravgPwr:=ravgPwr+Pbinr[j];
          iavgPwr:=iavgPwr+Pbini[j];
      (*     rgain_mu[j]:=Misadjust/(Pbinr[j]+1.0E-8);    *)
      (*     igain_mu[j]:=Misadjust/(Pbini[j]+1.0E-8);    *)
      (*     WriteLn(Lst, ' rmu(',j,') =', rgain_mu[j]);  *)
      (*     WriteLn(Lst, ' imu(',j,') =', igain_mu[j]);  *)
        End;
      avgPwr:=(ravgPwr+iavgPwr)*(1/(4*N));
      WriteLn(Lst, '  avgPwr = ',avgPwr);
      mu:=Misadjust/avgPwr;
      WriteLn(Lst, '  gain constant mu = ', mu);
    End;


(*************************************************************)
(*                  Procedure Conjugate_X                    *)
(*                                                           *)
(*   This procedure creates the conjugate of the diagonal    *)
(*   matrix X. The routine creates the conjugate of X        *)
(*   by creating XI_conj[j] which is the negative of XI[j].  *)
(*   Then, the conjugate of the diagonal of X equals         *)
(*   XR+XI_conj.                                             *)
(*                                                           *)
(*     Called By:                                            *)
(*        Main Program                                       *)
(*     Routines Called: None                                 *)
(*     Variables:                                            *)
(*       XI_conj[j] : -XI[j]                                 *)
(*                                                           *)
(*************************************************************)
```

```
Procedure Conjugate_X;
  Var
    j:integer;

       Begin

          For j:=1 to 2*N Do
             XI_conj[j]:=-XI[j];

       End;


(*****************************************************************)
(*          Procedure Perform_Matrix_Multiply                    *)
(*                                                               *)
(*    Multiplies a 2Nx2N matrix by a 2Nx1 dimension vector.      *)
(*    The matrix in all cases is a diagonal matrix so the        *)
(*    routine automatically ignores the off diagonal terms       *)
(*    during the multiplication.                                 *)
(*                                                               *)
(*     Called By:                                                *)
(*       Procedure Calculate_y                                   *)
(*       Procedure Calculate_Gradient                            *)
(*     Routines Called: None                                     *)
(*     Variables:                                                *)
(*       V2I[j] : the resulting vector imaginary component       *)
(*       V2R[j] : the resulting vector real component            *)
(*                                                               *)
(*****************************************************************)

Procedure Perform_Matrix_Multiply(var MR,MI,VR,VI: RealArrayN2);


  Var
     j:integer;
     dR1,dR2,dI1,dI2:double,
     RR1,RR2,RI1,RI2:real;

       Begin

           For j:=1 to 2*N Do
               begin
                  dR1:=MR[j]*VR[j];
                  dR2:=-(MI[j]*VI[j]);
                  RR1:=dR1;
```

```
                RR2:=dR2;
                V2R[j]:=RR1+RR2;
                dI1:=MR[j]*VI[j];
                dI2:=MI[j]*VR[j];
                RI1:=dI1;
                RI2:=dI2;
                V2I[j]:=RI1+RI2;
              end;

      End;

(*************************************************************)
(*                Procedure Calculate_y                      *)
(*                                                           *)
(*  This procedure calculates the output sequence values     *)
(*  for the current block being processed.  The output is    *)
(*  equal to the last N terms of the inverse FFT of the      *)
(*  product of X and the complex weight vector.              *)
(*                                                           *)
(*     Called By:                                            *)
(*       Main Program                                        *)
(*     Routines Called:                                      *)
(*       Procedure FFT                                       *)
(*       Procedure Prepare_input_block_for_FFT               *)
(*       Procedure Perform_Matrix_Multiply                   *)
(*     Variables:                                            *)
(*       y[j] : filter output                                *)
(*                                                           *)
(*************************************************************)

Procedure Calculate_y;
  Var
   j:integer;

   Begin

     Perform_Matrix_Multiply(XR,XI,HR,HI);
     isign:=-1;
     Prepare_input_block_for_FFT;
     FFT;
     For j:=1 to N Do
     y[k*N+j]:=(1/(2*N))*data[2*(N+j)-1];

   End;
```

```
(*************************************************************)
(*               Procedure Calculate_error                   *)
(*                                                           *)
(*  This procedure calculates the error sequence values for  *)
(*  the current block and the complex error vector.          *)
(*  The error block for the current input                    *)
(*  block equals the current desired block minus the output  *)
(*  for the current input block.  The complex error vector   *)
(*  E[j], equals the FFT of the zero padded error block:     *)
(*  FFT[N zeros, error block].                               *)
(*                                                           *)
(*    Called By:                                             *)
(*      Main Program                                         *)
(*    Routines Called:                                       *)
(*      Procedure Prepare_input_block_for_FFT                *)
(*      Procedure FFT                                        *)
(*    Variables:                                             *)
(*        ER[j]  :  real part of the FFT of the error sequence *)
(*                  for the current block                    *)
(*        EI[j]  :  imaginary part of the FFT of the error seq *)
(*                  for the current block                    *)
(*                                                           *)
(*************************************************************)


Procedure Calculate_error;
   Var
     j:integer;
     e:RealArrayN2;

      Begin

        For j:=1 to N Do
           begin
               e[j]:=0;
               input_block[j]:=e[j];
           end;
        For j:=N+1 to 2*N Do
            Begin
               e[j]:=desired[k*N+j-N]-y[k*N+j-N];
               input_block[j]:=e[j];
               error[k*N+j-N]:=e[j];
            End;
        isign:=1;
        Prepare_input_block_for_FFT;
```

```
        FFT;
        For j:=1 to 2*N Do
            begin
               ER[j]:=data[2*j-1];
               EI[j]:=data[2*j];
            end;

    End;

(*************************************************************)
(*              Procedure Calculate_Gradient                 *)
(*                                                           *)
(* This procedure calculates the gradient sequence for the   *)
(*  current block being processed. The gradient sequence     *)
(*  equals the first N terms of the inverse FFT of the       *)
(*  product of X conjugate and the error vector E.           *)
(*                                                           *)
(*   Called By:                                              *)
(*       Main Program                                        *)
(*   Routines Called:                                        *)
(*     Procedure Perform_Matrix_Multiply                     *)
(*     Procedure Prepare_input_block_for_FFT                 *)
(*     Procedure FFT                                         *)
(*   Variables:                                              *)
(*     grad[j]: gradient sequence for current block          *)
(*                                                           *)
(*************************************************************)


Procedure Calculate_Gradient;

  Var
     j:integer;

      Begin

          Perform_Matrix_Multiply(XR,XI_conj,ER,EI);
          isign:=-1;                        (* calc inverse *)
          Prepare_input_block_for_FFT;      (* FFT of X_conj*E *)
          FFT;
          For j:=1 to N Do
            grad[j]:=1/(2*N)*data[2*j-1];  (* grad = 1st N terms*)
          For j:=N+1 to 2*N Do              (* of inv FFT        *)
            grad[j]:=0;
```

```
     End;


(*************************************************************)
(*                 Procedure Update_weights                  *)
(*                                                           *)
(*   This procedure updates the filters complex weights. The *)
(*   new weights equal the old weights plus the product of the *)
(*   gain constant and the complex gradient vector.  The     *)
(*   complex gradient vector equals the FFT of the gradient  *)
(*   sequence padded with N zeros: FFT[(grad seq), N zeros].  *)
(*                                                           *)
(*     Called By:                                            *)
(*        Main Program                                       *)
(*     Routines Called:                                      *)
(*       Procedure Prepare_input_block_for_FFT               *)
(*       Procedure FFT                                       *)
(*     Variables:                                            *)
(*       HR[j] : real part of complex weight vector          *)
(*       HI[j] : imaginary part of complex weight vector     *)
(*************************************************************)


Procedure Update_weights;

     Var
        tap,j:integer;

          Begin

             For j:=1 to 2*N Do
                input_block[j]:=grad[j];
             isign:=1;
             Prepare_input_block_for_FFT;
             FFT;
             For tap:=1 to 2*N Do
                begin
                   HR[tap]:=HR[tap]+mu*data[2*tap-1];
                   HI[tap]:=HI[tap]+mu*data[2*tap];
                end;

          End;
(*************************************************************)
(*              Procedure time_domain_wts                    *)
(*                                                           *)
```

```
(*************************************************************)
Procedure time_domain_wts;
  Var
    j:integer;
    Begin
      For j:=1 to 2*N Do
        Begin
           V2R[j]:=HR[j];
           V2I[j]:=HI[j];
      (* WriteLn(Lst, '  H(', j,')= ',HR[j],'+i ',HI[j]); *)

        End;
      isign:=-1;
      Prepare_input_block_for_FFT;
      FFT;
 (*   WriteLn(Lst, 'Block ',k); *)
      For j:=1 to 2*N Do
         Begin
           WR[j]:=(1/(2*N))*data[2*j-1];
           WI[j]:=(1/(2*N))*data[2*j];
         End;
    End;
(*****************************************************************)
(*********************** Main Program ************************)
(*****************************************************************)


 Begin
    Open_input_files;
    Open_output_files;
    Init_var;
    Load_input;
    Load_desired;
    ClockOn;      (* Turn clock on *)
    Calc_numblocks;
    Calculate_avg_input_bin_pwr;
    Calculate_mu;
    For k:=0 to Block_num-1 Do
      Begin
       WriteLn( '  processing block',  k);
       Past_current_block;
       Diagonal_of_X;
       Calculate_y;
       Calculate_error;
       Conjugate_X;
```

```
        Calculate_gradient;
        Update_weights;
        time_domain_wts;
       end;
(*  ClockOff;  *) (*Display elapsed time*)
(* WriteLn;    *)
(* Write('Press Enter...'); *)
(* ReadLn;                  *)
(* time_domain_wts;  *)

    Write_output;
    Close_input_files;
    Close_output_files;

 End.
```

F.4 TDF Filter Listing

This is the Turbo Pascal 6.0 listing of the TDF program.

```
Program TDF;
 {$N+}
 Uses Printer,CRT,DOS;

 CONST
   datasize=992;
   N=16;
   Misadjust=0.1;
 TYPE
   Realarray=ARRAY[0..datasize-1] of real;
   RealarrayN=ARRAY[0..N-1] of real;

 VAR

   k:Integer;
   h,m,s,s100:Word;
   Mu,Px,StartClock,StopClock:Real;
   errorfile,outfile,infile,desiredfile:Text;
   error,desired,input,output:Realarray;
   weight_array:RealarrayN;
   weight0,weight1,weight2,weight3,weight4,weight5,weight6,weight7:text;



 (*************************************************************************)
 (***************** Initialize Variables  *********************************)
 (*************************************************************************)
 Procedure Init_Var;
    Var
       j:integer;
         Begin
           For j:=0 to datasize-1 Do
              Begin

                  desired[j]:=0.0;
                  error[j]:=0.0;
                  input[j]:=0.0;
                  output[j]:=0.0;
              End;
           For j:=0 to N-1 Do
```

```
          Weight_array[j]:=0;
      End;
  (*********************************************************)
  Procedure Open_input_files;
    Begin
      Assign(infile, 'A:S53N.Dat');
      Assign(desiredfile, 'A:S5.Dat');
      Reset(infile);
      Reset(desiredfile);
    End;
  (*********************************************************)
  Procedure Open_output_files;
    Begin
      Assign(errorfile, 'B:TFS3SNE.Dat');
      Rewrite(errorfile);
(*      Assign(outfile, 'B:TFS4SNY.Dat');   *)
(*      Rewrite(outfile);                   *)

    End;
  (*********************************************************)
  Procedure Close_input_files;
    Begin
       Close(infile);
       Close(desiredfile);
    End;
  (*********************************************************)
  Procedure Close_output_files;
    Begin
      Close(errorfile);
  (*    Close(outfile);    *)

    End;
  (*********************************************************)
  Procedure Load_input;
    Var
      j:integer;
      Begin
        For j:=0 to datasize-1 Do
          ReadLn(infile, input[j]);
      End;
  (*********************************************************)
  Procedure Load_desired;
    Var
      j:integer;
      Begin
```

```
      For j:=0 to datasize-1 Do
        ReadLn(desiredfile, desired[j]);
    End;
(*************************************************************)
Procedure ClockOn;
Begin
  GetTime(h,m,s,s100);
  StartClock:=(h*3600)+(m*60)+s+(s100/100);
End;
(*************************************************************)
Procedure ClockOff;
Begin
  GetTime(h,m,s,s100);
  StopClock:=(h*3600)+(m*60)+s+(s100/100);
  WriteLn('Elapsed time = ',(StopClock-StartClock):0:2);
End;
(*************************************************************)
Procedure Write_output;
  Var
    j:integer;
      Begin
        For j:=0 to datasize-1 Do
          Begin
       (*     WriteLn(outfile, output[j]);   *)
            WriteLn(errorfile,error[j]);
          End;
      End;
(*************************************************************)
Procedure Calculate_mu;
  Var
    j:integer;

    Begin
      Px:=0;
      For j:=0 to datasize-1 Do
        Px:=Px+Sqr(input[j])*(1/datasize);
      mu:=(1/N)*(1/Px)*Misadjust;
  (*    WriteLn(Lst, '  mu= ', mu);  *)
  (*    WriteLn(Lst, ' Signal Power = ',Px); *)
    End;

(*************************************************************)
Procedure Calculate_y;
   Var
     j:integer;
```

```
    sum_var:real;
  Label End_loop;
  Begin
     sum_var:=0;
     For j:=0 to N-1 Do
        Begin
         If (k-j<0) Then Goto End_loop;
         sum_var:=sum_var+weight_array[j]*input[k-j];
         End_loop:
        End;
     output[k]:=sum_var;
  End;
(*****************************************************************)
Procedure Calculate_error;
  Begin
    error[k]:=desired[k]-output[k];
  End;
(*****************************************************************)
Procedure Update_weights;
  Var
    j:integer;
  Label Skip_wt_update;
    Begin
      For j:=0 to N-1 Do
        Begin
          If (k-j<0) then Goto Skip_wt_update;
          weight_array[j]:=weight_array[j]+
                          2*mu*error[k]*input[k-j];
          Skip_wt_update:
        End;
    End;
(*****************************************************************)
Procedure Store_weights;
  Var
    j:integer;
    Begin
      For j:=0 to N-1 Do
        Begin
          If j=0 then
            WriteLn(Lst,weight_array[j]);
          If j=1 then
            WriteLn(Lst,weight_array[j]);
          If j=2 then
            WriteLn(Lst,weight_array[j]);
          If j=3 then
```

```
            WriteLn(Lst,weight_array[j]);
          If j=4 then
            WriteLn(Lst,weight_array[j]);
          If j=5 then
            WriteLn(Lst,weight_array[j]);
          If j=6 then
            WriteLn(Lst,weight_array[j]);
          If j=7 then
            WriteLn(Lst,weight_array[j]);
        End;
    End;

(*************************************************************************)
(***********************    Main Program    ************************)
(*************************************************************************)
  Begin
    Open_input_files;
    Open_output_files;
    Init_Var;
    Load_input;
    Load_desired;
(*    ClockOn;  *)
    Calculate_mu;

    For k:=0 to datasize-1 Do
       Begin
          Calculate_y;
          Calculate_error;
          Update_weights;
       (*   Store_weights; *)
       End;
 (*    Store_weights;   *)
(*     ClockOff;  *) (*Display elapsed time*)
(*     WriteLn;     *)
(*     Write('Press Enter...');    *)
(*     ReadLn;      *)
    Write_output;
    Close_input_files;
    Close_output_files;

  End.
```

Bibliography

1. Beauchamp, Kenneth G. *Applications of Walsh and Related Functions.* Orlando FL: Academic Press, Inc., 1984.
2. Cowan, Colin F.N. and Peter M. Grant. *Adaptive Filters.* Englewood Cliffs NJ: Prentice Hall, Inc., 1985.
3. Kepley, Capt Jeffery A. *Estimation of Evoked Fields Using a Time-Sequenced Adaptive Filter With The Modified P-Vector Algorithm.* MS thesis. AFIT/GE/ENG/90D-10. School of Engineering, Air Force Institute of Technology (AU). Wright-Patterson AFB OH, December 1990. (AD-A230474).
4. K.M.Wong and Y.G.Jan. "Adaptive Walsh Equalizer for Data Transmission." *IEE Proceedings. 130*(2):153-160 (March 1983).
5. Oppenheim, Alan V. and Ronald W. Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs NJ: Prentice Hall, Inc., 1989.
6. Press, William H. and others. *Numerical Recipes In Pascal.* New York NY: Cambridge University Press, 1989.
7. R.D.Brown. "A recursive algorithm for sequency-ordered fast Walsh transforms." *IEEE Transactions in Computing. C-26*:819-822 (1977).
8. VanCleave, James. *Walsh Preprocessor.* Final Report, 20 February 1979-31 March 1980 Contract F30602-79-C-0048. Lansdale PA: American Electronic Laboratories, Inc., August 1980. (AD-A091188).
9. Widrow, Bernard and Samuel D. Stearns. *Adapt... Signal Processing.* Englewood Cliffs NJ: Prentice Hall, Inc., 1985.
10. Williams, Capt Robert. "EENG 791 class lecture." School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1991.
11. Williams, Capt Robert. "Personal Interview on Spaced-based systems." School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, October 1991.
12. Williams, Robert. *Adaptive Filtering of Nonstationary Signals Using a Modified P-Vector Algorithm.* PhD dissertation. University of Dayton, Dayton OH, December 1989.